OMSI PASCAL-1 VERSION 1.2 MANUAL FOR RT-11

CONTENTS

INTRODUCTION

Welcome to Oregon Software's Pascal-1!

This user manual contains all the information you should need to install and operate Oregon Software's Pascal-1 system on DEC's RT-11 V2, V3 and V4 operating systems.

The first part of this manual that you will need is the Installation Guide, which is in the back. (We put it there because you will want it only for reference after you have installed your Pascal-1.) The other sections come in the order in which you are most likely to need them.

The first section is the User's Guide, which is a beginner's walk in the countryside of Pascal-1. The User's Guide has two parts. The first is a quick jump into simple programs. The idea is to immerse you immediately in our product, to give you a feel for it. The second part covers some of the same ideas, but expands on concepts and details. You can cover this section at a brisk pace. Experienced programmers will probably dash through it.

But lest you get too confident in your stroll through the User's Guide, there are brambles aplenty for unwary souls in the following sections: the Programmer's Guide, the Language Specification, and the Debugger Guide. Experienced programmers should have no trouble, and beginners can make their way carefully, but we encourage the latter toward additional reading. (See the list of books that follows.) Each section has a brief introduction explaining its function.

For More Information

We can suggest several places to find more information about Pascal:

(1) Try it! Certainly the most challenging course, and the most open-ended and accurate as well. Acquire the habit of answering your questions by experiment. Remember, "You can't hurt the computer!"

(2) Programming in Pascal, by Grogono -- a good course in Standard Pascal, with lots of sample programs for (1), above.

(3) Pascal User Manual and Report by Kathleen Jensen and Niklaus Wirth -- the first definition of Standard Pascal.

(4) This manual -- for fine points and grubby details of our Pascal, it's the only place in town.

For the serious student, these books are available from Oregon Software or elsewhere:

Systematic Programming: An Introduction, Niklaus Wirth;
Prentice-Hall, $17.75

Algorithms + Data Structures = Programs, Niklaus Wirth;
Prentice-Hall, $20.25

Structured Programming, Dahl, Dijkstra, Hoare;
Academic Press, $15.30

Elements of Programming Style, Kernighan and Plauger;
McGraw-Hill, $3.95

And we recommend that you join the Pascal Users' Group, which publishes an excellent newsletter. Send $10 for a one-year subscription to:

Pascal Users' Group
Attn: Rick Shaw
P.O. Box 888524
Atlanta, Georgia 30338
(404) 252-2600

Who Are We, Anyway?

Oregon Software traces its origins to OMSI -- the Oregon Museum of Science and Industry. OMSI is a private educational organization chartered "to further the education of the youth of the community", and it was in the Research Laboratory at OMSI that we began writing software. Seven of us came from OMSI to found Oregon Software in September, 1977. Because of the close association, the name "OMSI" stayed with us for a while, and we continue to contribute some of our corporate resources to the Museum.

But please, we're Oregon Software. We're a research and development software corporation in Portland, with a nice view of Mount Hood and of what's left of Mount St. Helens. The seven from OMSI have grown into twenty-five from all over.

On a serious note: OMSI is a non-profit, charitable institution. Contributions of money and equipment are much needed and are tax-deductible. Please earmark your donations for the Research Lab, which supports independent science projects in many fields, including computing. For further information about the OMSI Research Lab program, contact:

Director of Research
OMSI
4015 SW Canyon Road
Portland, Oregon 97201
(503) 248-5943

What is Pascal-2?

Now that you know more about us, you may want to know more about our newest product-in-progress, Pascal-2. Pascal-2 is our new optimizing compiler, written in Pascal. It's designed to be portable, and it's already been moved to a Honeywell computer. The Pascal-2 compiler is bigger and slower than Pascal-1, but not the generated code. Typical programs are 40% smaller and almost twice as fast. You can expect Pascal-2 compilers to be available on a wide range of 16-bit and 32-bit processors in the next several years. Supported users of Pascal-1 will receive substantial discounts on their purchase of Pascal-2 licenses for the PDP-11.

And Finally . . .

This manual is the third edition of our RT-11 manual and contains a number of changes and additions, including:

1) Correction of errors and omissions in documentation;

2) More details, especially about installation procedures;

3) A general Table of Contents and a Table of Contents for each section;

4) A standard format for all sections and sequential numbering for the entire manual;

5) Inclusion of all sample programs in the machine-readable version of the manual;

6) Correction of spelling mistakes and other irritating little errors.

Most of the changes are in the nature of "tidying up", but we believe that it is a good deal easier to read and find things in this version of our manual than in our earlier efforts.

Oregon Software plans to continually upgrade its written materials. In the future, we expect to include more details and more sample programs and examples in the RT-11 manual. If you have any suggestions, or if you have any problems with this manual, please tell us about them (in writing).

SECTION 1: USER'S GUIDE

Introduction to the User's Guide

This is the introductory section, the User's Guide. It explains:

1) how to compile and run your Pascal programs;

2) how to interpret program listings and error messages;

3) some details of the compilation process.

This guide assumes that you are familiar with:

1) simple RT-11 commands;

2) a text editor (EDIT, TECO, EDT, SOS);

3) elementary Pascal programming.

This guide is not:

1) an introduction to Pascal (see Programming in Pascal by Grogono);

2) a detailed description of Pascal-1 (see the Language Specification);

3) an expert's guide to Pascal-1 (see the Programmer's Guide).

In examples, underlining is used to show the text that you should type. Non-underlined text shows prompts or other responses by the computer.

Entering the Program

So you want to run a Pascal program?

The first step is to enter the program into the computer and store it in the file system. Use a familiar text editor to enter your program, and store the program in a file with the extension .PAS. The Pascal system accepts free-format program files, so use blanks, tabs, new lines, and form feeds as desired to help make your program readable.

This Pascal version of your program is called the source program, or the source file. All other versions of your program are translations from the source program.

Source programs should be stored in files with the extension .PAS for Pascal (example: FIRST.PAS). The .PAS extension may be omitted from commands to the Pascal system, but must be included in commands to other RT-11 systems such as the editor.

Compiling the Program

After editing, you must compile your program -- translate it into a form that can be directly executed by the computer. The Pascal compilation process is directed by the PCL (Pascal Command Language) program. Its simplest form is:

```
.R PCL
*<source file name>
```

To illustrate the compilation process, let's assume that this program

```
program First (output);
begin
   write ('"Things are best in their beginnings"');
   writeln (' -- Blaise Pascal');
end.
```

is stored in the file FIRST.PAS.

The compilation process begins with your command and is followed by the computer's response:

```
.R PCL
*FIRST

.RU SY:PASCAL
*TEMP.TMP,TT:=FIRST/N

Errors detected:  0
Free memory: 11660 words

.R MACRO
*TEMP.TMP=TEMP.TMP
ERRORS DETECTED:  0
*^C

.R LINK
*FIRST=TEMP.TMP,SY:PASCAL
*^C

.RU FIRST
"Things are best in their beginnings" -- Blaise Pascal
```

Notice that the PCL command line compiles the source program and then immediately runs (executes) the compiled program. This is known as "compile and go" operation. Compiled programs are stored in files with the .SAV extension (FIRST.SAV). After being compiled, the .SAV program can be executed again with the 'R' or 'RUN' commands.

Notice also that no errors were detected. This is certainly unusual if this is your first program! What happens if there are detectable errors in the source program? The next example shows the result of such an error.

Correcting the Program

The following program contains a deliberate error:

```
program Second (output)
begin
   writeln ('Things get worse as they continue');
end.
```

This program is missing a semicolon between the program heading and the keyword 'begin'. Semicolon errors are the most common errors made by beginning Pascal programmers. Semicolon errors are always detected by the compiler:

```
.R PCL
*SECOND

.RU SY:PASCAL
*TEMP.TMP,TT:=SECOND/N
  2       BEGIN
             ^
******  Expected 'SEMICOLON' missing

Errors detected: 1
Free memory: 11679 words

.R MACRO
*TEMP.TMP=TEMP.TMP
^C
.
```

For each error, a line of the source program is printed, then an arrow indicating the approximate position of the error, and a message describing the error. Many compilation errors are possible. See Appendix A of the Programmer's Guide for a complete list.

The Program Listing (/L)

Often, we need to see more of the program to locate and correct the error. The Pascal compiler can be directed to display the entire program, with all detected errors and other information. This is the 'listing' of the program. To get a program listing at your terminal, add the /L switch to the PCL command line:

```
.R PCL
*<source file name>/L
```

To print a listing to the line printer or another destination, give a comma followed by the destination file name, an equal sign, and the source file:

```
.R PCL
* , <listing file name> = <source file name>
```

Note: listing files are created with the .LST extension. A listing of a sample program follows:

```
THIRD  OMSI Pascal V1.2G RT11 26-Dec-80 08:29 Site #1-1 Page 1
Oregon Software  Portland, Oregon 97201  (503) 226-7760

Line  Stmt  Level  Nest                    Source Program

    1                      program Third (output)
    2                      begin
                           ^
******  Expected 'SEMICOLON' missing

    3    1       1     1   writeln ('Things get hazy if you stare');
    4    2       1     1   end.

Errors detected:  1
Free memory: 11677  words
```

The listing is printed in pages, with a heading on each page showing the program name, the exact version of the Pascal system, the date and time, and the licensed site identification (the facility name and site number).

Four columns of numbers appear on the left side of each page. The first column, labeled "Line", simply numbers each line of the source program. The second column is labeled "Stmt" and gives the statement number of the first statement on that line. The statement number starts at 1 for each control section, and increases by one as each statement is compiled. An up-to-date listing can be useful during debugging, because the statement numbers are used by the Debugger to identify breakpoints.

Listing for a Complex Program

To illustrate the "Level" and "Nest" columns, a more complex program is needed:

```
FINAL  OMSI Pascal V1.2G RT11 26-Dec-80 09:30 Site #1-1  Page 1
Oregon Software  Portland, Oregon 97201 (503) 226-7760

Line  Stmt Level  Nest   Source program

   1                     program Final (output);
   2
   3                     const Reality = true;
   4
   5                     procedure Objective;
   6                     begin
   7     1     2     1     if Reality then
   8     2     2     2       write('Things become more complex ')
   9     3     2     2     else write('In the Beginning, ...');
  10     4     2     1   end;
  11
  12                     procedure Awareness;
  13                     var Eye: (Subject, Object);
  14                     begin
  15     1     2     1     for Eye := Subject to Subject do
  16     2     2     2       writeln('as one understands them');
  17     3     2     1   end;
  18
  19                     begin
  20     1     1     1     Objective;
  21     2     1     1     Awareness;
  22     3     1     1   end.


Errors detected:  0
Free memory:  11554  words
```

The "Level" column shows the depth of procedure nesting. The main program is at level 1, its procedures are level 2, and so on; a procedure at level 4 is enclosed by two surrounding procedures or functions. The "Nest" column shows a similar nesting of statements within other structured statements.

More on PCL

The PCL command line can include up to six source files combined in sequence to form the complete program. The compilation also can be modified by "switches" in the PCL command line. A switch is a slash followed by a letter (such as /L, which prints the listing). The most commonly used switches are shown in the following examples. The Programmer's Guide has a complete list.

To further demonstrate the use of the PCL, let's compile and list the program E. This program calculates an approximation of E (the base of the natural logarithms) by summing the series

    1+1/1!+1/2!+1/3!+ ... +1/N!

until additional terms do not affect the approximation.

```
.R PCL
*E/L

.RU SY:PASCAL
*TEMP.TMP,TT:=E

EX    OMSI Pascal V1.2G RT11 26-Dec-80 12:58 Site #1-1   Page 1
Oregon Software  Portland, Oregon 97201 (503) 226-7760

Line  Stmt Level Nest   Source Statement

   1                    program EX;
   2                    var E, Delta, Fact: real;
   3                        N: integer;
   4                    begin
   5     1     1    1     E:= 1.0; Fact:= 1.0; Delta:= 1.0;
   6     4     1    1     N:= 1;
   7     5     1    1     repeat
   8     6     1    2       E:= E + Delta;
   9     7     1    2       N:= N + 1; Fact:= Fact * N;
  10     9     1    2       Delta:= 1 / Fact;
  11    10     1    2     until E = (E + Delta);
  12    11     1    1     write('With ', N:1, ' terms, ');
  13    12     1    1     writeln('the value of E is', E:18:15);
  14    13     1    1   end.

Errors detected:  0
Free memory: 11636  words

Errors detected:  0
Free memory: 11636  words

.R MACRO
*TEMP.TMP=TEMP.TMP
ERRORS DETECTED:  0
*^C

.R LINK
*E=TEMP.TMP,SY:PASCAL
*^C

.RU E
With 11 terms, the value of E is 2.718280000000000
```

The Debugger (/D/S)

We can watch the progress of the computation and can display intermediate values without making any program changes. For this, we use the /D/S switch pair to compile with the interactive Debugger. After compiling, we set a stored breakpoint command to display the current value of E.

```
.R PCL
*E/D/S

.RU SY:PASCAL
*TEMP.TMP,E,E=E/D/S

Errors detected:  0
Free memory:  11372  words

.R MACRO
*TEMP.TMP=TEMP.TMP
ERRORS DETECTED:  0
*^C

.R LINK
*E=TEMP.TMP,SY:PASCAL/C/T
*SY:A/O:1/C
*SY:B/O:1/C
*SY:0/O:2/C
*SY:1/O:2/C
*SY:2/O:2/C
*SY:3/O:2/C
*SY:4/O:2/C
*SY:5/O:2/C
*SY:6/O:2/C
*SY:7/O:2/C
*SY:8/O:2/C
*SY:9/O:2
Transfer symbol? $START
*^C
```

```
.R E
```

At this point, the Debugger will prompt you for the name of your program, and you can set the breakpoints. In this example, we set the breakpoint at MAIN,6 because this is the point at which the value of E changes in this particular program (see the listing on Page 10). We also tell the program to write the value of E at that point and then continue. (See the Debugger Guide for details of these commands.)

```
POD (Pascal Online Debugger) - 24-Apr-79
POD - Program name? E
} B(MAIN,6) <W(E);C>
} G
Breakpoint at MAIN,6  E:= E + Delta;
1.000000
Breakpoint at MAIN,6  E:= E + Delta;
2.000000
Breakpoint at MAIN,6  E:= E + Delta;
2.500000
Breakpoint at MAIN,6  E:= E + Delta;
2.666667
Breakpoint at MAIN,6  E:= E + Delta;
2.708333
Breakpoint at MAIN,6  E:= E + Delta;
2.716667
Breakpoint at MAIN,6  E:= E + Delta;
2.718056
Breakpoint at MAIN,6  E:= E + Delta;
2.718254
Breakpoint at MAIN,6  E:= E + Delta;
2.718279
Breakpoint at MAIN,6  E:= E + Delta;
2.718282
With 11 terms, the value of E is 2.718280000000000
Program terminated at MAIN,12  end.
}^C
```

Extended Precision (/X)

The computed value is printed with 6 significant digits. For more precision, we can use the /X switch, which means "extended precision". With extended precision, 15 significant digits are computed and displayed. See the Programmer's Guide.

```
.R PCL
*E/X

.RU SY:PASCAL
*TEMP.TMP,TT:=E/X/N

Errors detected:  0
Free memory:  11636   words


.R MACRO
*TEMP.TMP=TEMP.TMP
ERRORS DETECTED:  0
*^C

.R LINK
*E=TEMP.SY:PASCAL
*^C

.RU E
With 19 terms, the value of E is 2.718281828459050

.
```

The Profiler (/P/S)

Finally, let's "profile" the program by using the /P/S combination. The profile listing shows exactly the number of times each line is executed, allowing us to concentrate on the parts of the program that might be optimized effectively.

```
.R PCL
*E/P/S

.RU SY:PASCAL
*TEMP.TMP,E,E=E/D/S

Errors detected:  0
Free memory: 11372  words


.R MACRO
TEMP.TMP=TEMP.TMP
ERRORS DETECTED: 0
*^C

.R LINK
*E=TEMP.TMP,SY:PASCAL,SY:PROFIL
*^C

.RU E
Program name? E
Output profile to: TI:
With 11 terms, the value of E is 2.718280000000000

EX    OMSI Pascal V1.2G RT11 26-Dec-80 12:59 Site #1-1   Page 1
Oregon Software  Portland, Oregon 97201 (503) 226-7760

     Line  Stmt Level Nest   Source Statement

        1                    program EX;
        2                    var E, Delta, Fact: real;
        3                        N: integer;
        4                    begin
  1     5     1     1     1     E:= 1.0; Fact:= 1.0; Delta:= 1.0;
  1     6     4     1     1     N:= 1;
  1     7     5     1     1     repeat
 10     8     6     1     2       E:= E + Delta;
 10     9     7     1     2       N:= N + 1; Fact:= Fact * N;
 10    10     9     1     2       Delta:= 1 / Fact;
 10    11    10     1     2     until E = (E + Delta);
  1    12    11     1     1     write('With ', N:1, ' terms, ');
  1    13    12     1     1     writeln('E is', E:18:15);
  1    14    13     1     1   end.

Errors detected:  0
Free memory: 11636  words
```

Your Next Step

Thus ends your guided tour through Pascal-1. At this point, you should be able to run a few simple programs. Before getting into complex programs, however, you should consult the Programmer's Guide, the Language Specification, and the Debugger Guide.

SECTION 2: PROGRAMMER'S GUIDE

Introduction to the Programmer's Guide

The Programmer's Guide describes the way in which the Pascal-1 V1.2 system interacts with the PDP-11 and the way in which some of the advanced features of Pascal-1 operate.

This guide assumes that you are experienced in programming with Pascal.

This guide is not:

1) an introduction to Pascal (see Programming in Pascal by Grogono);

2) a beginner's guide to Pascal-1 (see the User Guide);

3) a detailed description of Pascal-1 (see the Language Specification);

In examples, underlining is used to show the text that you should type. Non-underlined text shows the prompts or other responses by the computer.

Compilation Switches

The compilation process and the resulting program can be modified by switches appearing in the PCL command line. Switches are a single alphabetic character after the '/' (slash) marker, as in the command line MAIN/D/S.

The complete set of compilation switches appears below, followed by a detailed description of each switch.

/C	Compile only	Complete compilation but no execution
/D	Debug	Debugger compilation
/E	External	External module -- implies /O
/F	Fast reals	Generate calls rather than traps
/I	Improver	Branch/jump resolution
/L	Listing	Produce compilation listing
/L:n	Listing	Specify listing page length
/M	Macro	Partial compilation to assembler .MAC
/N	Nolist	List errors only
/O	Object	Partial compilation to linker .OBJ
/P	Profile	Profiler compilation
/Q	Quick	Uses fast MAC.SAV assembler
/S	Source	Include source lines (modifies /D/P/M)
/X	eXtend	Extended precision Reals (15 digits)

Listing Control Switches (/L, /L:n, /N)

The /L switch directs the compiler to produce a listing. The /L:n switch indicates that the listing is to be in pages of N lines each. The /N switch directs the compiler to list only lines in error. The /L and /N switches are related to the $L+ and $L- embedded switches.

Partial Compilation Switches (/C, /M, /O)

These switches interrupt the compilation process when intermediate results are desired.

The /M switch performs only the first compilation step, resulting in an assembly source translation of the Pascal program.

The assembler source file has the extension .MAC. The /S switch is recommended with /M and will include the Pascal source lines as comments.

The /O switch performs the compilation and assembly steps, producing a relocatable object file suitable for input to the Linker or Librarian. The /O switch is implied by the /E (External) switch.

The /C switch performs the entire compilation process, but does not execute the resulting program. The program may then be run with the R or RUN commands.

Real Arithmetic Switches (/X, /F)

The /X switch causes the compiler to use extended precison for values of type Real. All Real values are extended. It is not possible to mix normal and extended precision values. The /X switch is related to the $X embedded switch. See the section on Extended Precision.

The /F switch is of limited utility. On processors lacking both FPP and FIS floating-point hardware, Real operations are normally performed by a trap of each FIS instruction and simulation of its effects. The trapping process requires some overhead, but is compact. The /F switch causes the compiler to generate subroutine calls rather than simulating FIS instructions. The subroutine calls are faster but require an extra word for each floating-point instruction.

Debugger Switch (/D)

The /D switch indicates a Debugger compilation. This switch causes generation of a symbol table file and, if /S is also present, a listing file. The /D switch also causes generation of code to identify each procedure and statement to the interactive debugger. The /D switch is related to the $D+ and $D- embedded switches. See the section on the Debugger.

Profiler Switch (/P)

The /P switch causes the inclusion of code for performance measurement, and generates a symbol table file and, if /S is also specified, a listing file. The Profiler uses the Debugger interface code, so that /P cannot appear with /D. The /P switch is related to the $D+ and $D- embedded switches. See the section on the Profiler.

Source Mode Switch (/S)

The /S switch performs two distinct functions. When used with the Debugger (/D/S) or Profiler (/P/S), /S enables the source program mode of operation and connects the actions of the Profiler and Debugger to the source text of the program.

When used with the Macro switch (/M/S), the generated assembly language translation will include the Pascal source lines embedded as comments within the assembly file. This use of the /S switch is related to the $S+ and $S- embedded switches.

External Module Switch (/E)

The /E switch indicates an external module compilation. This causes the outermost procedures and functions to be identified to the Linker with global entry names. An external module can include global declarations, procedures, and functions but is not required to include a main control section. The /E switch is related to the $E embedded switch. See the External Module section. The /E switch should not be used to compile the main program that calls external procedures.

Branch/Jump Improver Switch (/I)

The /I switch inserts an additional step into the compilation process, a branch/jump resolver (IMP). IMP replaces branch/jump combinations with a single conditional branch where possible. The Improver runs slowly, but typically reduces program size by 6 to 8 percent. IMP is recommended only for compilation of fully debugged production programs.

Fast Assembler Switch (/Q)

The /Q switch causes the compilation process to use the MAC.SAV assembler instead of the MACRO assembler. MAC.SAV is a single-pass assembler that has no macro or local symbol capabilities, but it assembles compiler output in about one third of the time required by the RT-11 MACRO assembler.

Embedded Switches

Embedded switches provide control of compilation options within the Pascal source program. Embedded switches have the form of a Pascal comment beginning with a dollar sign ($), followed by a single uppercase alphabetic character and possibly a plus or minus sign, as in (*$L+*). Several of the embedded switch functions can also be provided by compilation switches. Embedded switches have the advantage that, once included in a program, they cannot be accidentally omitted from a compilation.

The complete list of embedded switches below is followed by a more detailed description of each switch function. In general, a plus sign enables a particular function, and a minus sign disables it. Switches that are initially enabled are marked with [+]; switches marked [MBF] ('must be first') must appear before any Pascal code.

$A-,$A+	Array bounds	Include array bounds check [+]
$C	Code insert	See the Embedded Assembly Code section
$D-,$D+	Debugger	Include debugger interface
$E-,$E+	External	External module compilation
$F-,$F+	Fast FIS	Enable floating-point calls
$L-,$L+	Listing	Source lines in listing [+]
$S-,$S+	Source mode	Source lines in assembly
$T-,$T+	sTack check	Include stack overflow check [+]
$X	eXtend	Extended precision reals [MBF]

Error-Checking Switches ($A, $T)

The $A switch controls the generation of code to check each array reference and ensure that the index is within the bounds of the array. Bounds checking is initially enabled; the $A- switch will disable checking. If enabled, each bounds check adds 8 words to the size of the program.

The $T switch controls stack overflow checking, and is initially enabled. Stack overflow is possible upon entry to any procedure or function block. This check can be disabled with $T-, with a small savings of memory (2 words per procedure).

Debugger/Profiler Switch ($D)

The $D switch controls the interface code to the Debugger and Profiler. If enabled, each statement and procedure includes instructions to call the Debugger or Profiler. These instructions require 1 to 3 words per statement (1 word for statements 1-255 of each procedure, 2 words otherwise, and an additional word if /S source mode is enabled). In large programs, one may choose to disable debug interface code generation for sections known to be correct.

External Module Switch ($E)

Enabling the $E switch causes global procedures and functions to be labeled as external entry points in the relocatable object file. A main program section is ignored if it is encountered when the $E switch is enabled. See the External Module section.

Real Arithmetic Mode Switches ($X, $F)

The $X switch enables extended precision (15-digit) real arithmetic. If present, the $X switch must precede any Pascal code. You cannot mix normal and extended precision in one program, so that each module in separate compilations must be compiled with the same precision. See the Extended Precision section.

The $F switch is useful only on processors lacking FIS and FPP hardware for floating point calculations. On these processors, floating-point instructions are normally trapped and simulated. The $F switch instead causes direct subroutine calls to floating-point routines, saving about 0.2 milliseconds per floating-point instruction at the cost of an extra word.

Listing Control Switch ($L)

The $L switch controls the appearance of lines in the program listing file. If enabled, all program text will appear in the listing. If the $L switch is disabled, only lines in error and error messages will appear.

Source Mode Switch ($S)

Enabling the $S switch causes the Pascal source lines to appear as comments in the compiler assembly output. This makes it easier to determine the code generated for each statement. This switch also modifies the behavior of the Debugger and Profiler.

I/O Control Switches

The Reset() and Rewrite() standard procedures accept additional arguments specifying a Filename of an external file, and a DefaultName with default fields of the filename. These arguments can also include I/O control switches, which give explicit control of the operating system interface details.

The I/O switches appear in the Filename or DefaultName parameters as in this example:

```
Rewrite(F,'data.dat/seek/span/size:12.');
```

A special device (TI:) also may appear in the Reset() and Rewrite() calls. The TI: device connects to the Pascal-1 terminal driver and is used in place of the TT: driver for interactive use.

A complete list of I/O switches appears below, followed by individual details. All switches may be abbreviated to the first two letters.

/buffersize:n	Allocate N bytes for buffer
/go	Allow programmed error handling
/odt	Single character terminal input
/nfs	Non-File-Structured access
/seek	Direct-access file
/size:n	File storage allocation
/span	Records span block boundaries
/temp	Temporary file

/BUFFERSIZE:n Switch

Pascal-1 normally allocates the minimum space required for a file buffer, which is usually 512 bytes but is dependent on device and file characteristics. More efficient I/O transfers can be performed at the cost of additional memory. The /Buffersize:n switch specifies the storage to be allocated to a file buffer. The size value is a decimal number if terminated with a period, otherwise octal.

/GO Switch

I/O transfer errors are normally fatal and cause immediate program termination. The /Go switch indicates that transfer errors on the specified file are non-fatal and allow program execution to continue. In using this switch, the programmer accepts responsibility for checking the RT-11 I/O status code after each I/O operation. The error code for the previous I/O transfer error

is available in the byte at address 52B.

/ODT Switch

The /ODT switch derives its name from the ODT Debugger (Octal Debugging Technique), which is driven by single character commands. The /ODT switch is used with keyboard files, and indicates that each character read from the file is to be processed immediately without any wait for a carriage return or other action character. The /ODT switch also disables the normal one character buffer effect of the Read() standard procedure.

The rubout and ctrl/U keyboard editing capabilities are not effective on terminals open with /ODT.

/NFS Switch

The /NFS switch is used when you desire non-file-structured access to a device that is normally file-structured, such as a disk device. Because direct access to such a device can destroy its directory structure, Pascal-1 prevents non-file-structured access unless the /NFS switch is used.

/SEEK Switch

The /Seek switch performs two functions: it enables the use of the direct-access Seek() procedure, and it permits both read and write access to the file variable so that records may be updated.

/SIZE:n Switch

The /Size switch used in the Rewrite() procedure specifies the space to be allocated for the file. The size of the file is given in blocks of 512 bytes, and is a decimal number if terminated by a period, and octal otherwise.

/SPAN Switch

In files created or accessed by Pascal-1 programs, fixed-length records are normally 'blocked'. This means that an integral number of records are stored in one disk block of 512 bytes, with any remaining storage in that block being unused. The /Span switch packs records more efficiently, with records spanning from one disk block to the next. This requires additional buffer memory, which is automatically allocated, and some additional computation. Spanned and blocked files are not generally compatible. Files created with /Span should be read with the same switch.

/TEMP Switch

This switch is used in Rewrite() to indicate a temporary file that will be deleted on termination. No filename is needed if this switch appears.

The Profiler

The Profiler identifies the sections of a program that can be most effectively optimized. Empirical measurements show that typical programs consume a large fraction of their computation time in a small portion of the program code ("90% of the time in 10% of the code"). The Profiler counts the actual number of times each statement is executed and each procedure is activated, and displays this information either in the program listing or in a tabular form.

The /P switch appears in the compilation command to invoke the Profiler. The /S switch is recommended in addition for more convenient display of the profile information.

When the Profiler begins executing, it will ask for the program name. The Profiler uses the symbol table and listing files produced by the compiler to identify procedures and statements in the program. The symbol table file normally has the same name as the program and the extension .SYM, and the listing file normally has the extension .LST. The Profiler will ask for the correct filenames if the normal files are not available.

The Profiler will then ask for the desired destination of the profile information. The profile will be written to the specified file with the default extension .PRO. This should be a permanent file (disk or hard copy device), as the Profiler requires roughly a factor of fifty performance overhead while gathering information.

The program being measured will then execute normally, although somewhat more slowly. Upon normal termination, or any fatal error, or ctrl/C interrupt, the profile information will be written to the specified file.

The first section of the profile is the Procedure Reference Profile, which lists each referenced procedure and function with the count of calls on that procedure. The second section is the Statement Reference Profile, displayed in tabular format. If the /S (source) switch is specified, this section displays the program listing with an additional column containing the reference count for each line.

The Profiler is limited in several respects: only the first 100 statements in each procedure will be counted, and a maximum of 40 procedures and functions can be profiled. The $D- and $D+ embedded switches can be used to selectively enable and disable profiling.

Example:

```
PRIMES  OMSI Pascal V1.2G RT11 26-Dec-80  8:14 Site #1-1  Page 1
Oregon Software  Portland, Oregon 97201 (503) 226-7760

      Line  Stmt Level Nest   Source Statement

         1                    program Primes; (* Author: N. Wirth *)
         2                    const N=2500; (* first 2500 Primes *)
         3                    type Index=1..N;
         4                    var X,Square: integer;
         5                     I,K,Lim: Index;
         6                     Prime: Boolean;
         7                     P: array[Index] of integer;
         8                     V: array[1..100] of integer;
         9                    begin
    1    10     1     1    1     P[1]:=2; X:=1; Lim:=1; Square:=4;
    1    11     5     1    1     write(2);
    1    12     6     1    1     for I:=2 to N do begin
 2499    13     8     1    3       repeat
11153    14     9     1    4         X:=X+2;
11153    15    10     1    4         if Square<=X then begin
   35    16    12     1    6           V[Lim]:=Square;
   35    17    13     1    6           Lim:=Lim+1;
   35    18    14     1    6           Square:=P[Lim]*P[Lim];
   35    19    15     1    6           end;
11153    20    16     1    4         K:=2; Prime:=true;
11153    21    18     1    4         while Prime and (K<Lim) do begin
94012    22    20     1    6           if V[K]<X
28669    23    21     1    7             then V[K]:=V[K]+P[K];
94012    24    22     1    6           Prime:=(X<>V[K]); K:=K+1;
94012    25    24     1    6           end;
11153    26    25     1    4       until Prime;
 2499    27    26     1    3       P[I]:=X; write(X);
 2499    28    28     1    3     end;
    1    29    29     1    1   end.
```

Format and Cross-Reference (PASFMT)

The PASFMT utility supplied with Pascal-1 will automatically reformat a Pascal source program, adjusting indention and partitioning statements so that a program listing reflects the program structure. The PASFMT program can also provide a cross-reference index of a Pascal source program showing block calls, nesting, and identifier references.

The PASFMT command line can contain one or two output files, an input source file, and several optional switches. Run PASFMT as follows:

```
.R PASFMT
PASFMT V2.0 (10Dec79)
*<Format>,<Crossref>=<Source>/switches
```

The Format output file is the formatted source program. Several switches select token translation options:

/L	Lowercase	Lowercase identifiers, uppercase keywords
/M	Mixedcase	Unchanged identifiers, uppercase keywords
/U	Uppercase	All letters uppercase

The Crossref output file (if specified) normally contains the program listing with line and page numbers, followed by the procedure call and nesting index. Two switch options apply to the cross reference:

/C	Crossref all	Cross reference all identifiers
/N	No listing	Produce only crossref index

The /C switch may be used only for source programs of moderate size, because of memory limitations.

The Improver (IMP)

The utility program IMP decreases the size of the object code produced by Pascal-1 by replacing branch/jump combinations with single branches when possible. IMP will reduce the generated code by roughly 5 to 8 percent.

IMP is included in the compilation process by the /I compilation switch, as in the command line TEST/I.

Note that IMP runs quite slowly and therefore is recommended for use only on completely debugged production programs.

Dynamic String Package

A package of procedures and functions for dynamic string processing is supplied with Pascal-1 V1.2, and is stored in the file STRING.PAS. The package is written in Standard Pascal, and allows programs using strings to be moved to other Pascal implementations.

Strings are stored as a record structure with a fixed maximum number of characters (normally 100 but easily changed), and an integer marking the current length of the string.

```
type String = record
        Len: Integer;
        Ch:  packed array[1..StringMax] of Char;
        end;
```

The capabilities provided are:

Len(S) -- returns the current length of string S;

Clear(S) -- initializes string S to empty;

ReadString(F,S) -- reads a value for string S from the text file F. The string is terminated by Eoln(F) and a Readln(F) is performed. String overflow (a string longer than StringMax) results in truncation.

WriteString(F,S) -- writes the string S to the text file F. The same effect can be achieved by passing the parameter S.Ch:S.Len to Write(), as in Write(F,'S=',S.Ch:S.Len).

Concatenate(T,S) -- appends string S to the target string T. The resulting value is string T. Overflow results in truncation to StringMax characters.

Search(T,S,Start) -- searchs string T for the first occurrence of string S to the right of position Start (characters are numbered beginning with one). The function Search() returns the position of the first character in the matching substring, or the value zero if the string S does not appear.

Insert(T,S,Start) -- inserts the string S into the target string T at position Start. Characters are shifted to the right as necessary. Overflow produces a truncated target string. A Start position that would produce a non-contiguous string has no effect.

The Start and Span parameters in the Substring and Delete procedures define a substring beginning at position Start (between characters Start-1 and Start) with a length of Abs(Span). If Span is positive, the substring is to the right of Start; if negative, the substring is to the left.

Delete(S,Start,Span) -- deletes the substring defined by Start, Span from the string S.

Substring(T,S,Start,Span) -- the substring of string S defined by Start, Span is assigned to the target string T.

An example of the use of the string package is given below. The example reads a line from the terminal and separates it into single words. The compilation command to PCL should be "STRING,EXAMPL".

```
type Lit = packed array [1..10] of char;

procedure Literal(var T: String; Ch: Lit; N: integer);
var I: integer;
begin
  Clear(T);
  for I := 1 to N do T.Ch[I] := Ch[I];
  T.Len := N;
end;

var Space, Line, Word: String;
    Mark: integer;
begin
  Literal(Space,'          ', 1);     { make 1 char string }
  write('Type a line: ');
  ReadString(Input, Line);
  Concatenate(Line, Space);
  Mark := Search(Line, Space, 1);
  while Mark > 0 do begin
    Substring(Word, Line, 1, Mark - 1);
    if Len(Word) > 0 then begin
      WriteString(Output,Word);
      writeln;
      end;
    Delete(Line, 1, Mark);
    Mark := Search(Line,'Space, 1);
    end;
end.
```

External Modules

External modules allow several program sections, each containing at least one procedure, function, or main program, to be compiled independently and combined at link time. External modules may be combined into module libraries to simplify handling of common routines. The external module interface also allows inclusion of modules written in other languages, such as FORTRAN and MACRO.

The EXTERNAL directive is used to reference a procedure or function in an external module. The declaration of an external procedure or function contains the procedure or function name and parameters, followed by the directive EXTERNAL (similar to FORWARD). The procedure or function body does not appear in the program unit referencing the external routine.

The FORTRAN directive replaces EXTERNAL to reference external routines written in FORTRAN or MACRO. The FORTRAN directive causes the generation of a PDP-11 standard calling sequence (the Pascal calling sequence places parameters on the stack, while the FORTRAN sequence points R5 to a list of parameters).

The /E compilation switch and the $E embedded switch are used to create modules that can be referenced by EXTERNAL directives. When the $E switch is enabled, each global procedure and function declaration causes an external (global) symbol to be defined. These global symbols are matched at link time to the global references created by the EXTERNAL directive.

The external reference symbols are composed of the first six characters of the external procedure or function identifier, and must uniquely identify the external routine. Duplication or overlap of external symbols results in the Link error 'Multiple definition', while a missing module results in the 'Undefined global' error message.

Several cautions should be observed when you use EXTERNAL and FORTRAN directives. Parameters to external modules cannot be checked by the compiler for type conformance, so an accidental type mismatch may cause entirely unpredictable results. The FORTRAN directive causes only generation of the proper calling sequence; the directive does not link or initialize the FORTRAN I/O system.

External modules may reference global (static) variables, which are shared by all of the modules composing a program. If all modules (including the main program) are compiled with the same global variables, the effect is as if all modules were compiled together. The compiler cannot verify type conformance of global data.

When combining modules to form libraries, remember that all procedures and functions from a compilation form a single module,

and cannot be individually selected from the library. The module name is taken from the first six characters of the program identifier (in the program heading).

The Linker, Librarian, and Overlays

Object modules produced by the Pascal compiler using the /O or /E compilation switches are compatible with object modules produced by the MACRO assembler, FORTRAN compiler, and other RT-11 system utility programs. The Linker (LINK) can be used to produce overlaid executable programs, allowing much larger programs. The Librarian (LIBR) is used to build libraries of object modules for more convenient handling. Some highlights of Linker and Librarian capabilities are covered here. See the RT-11 System User's Guide for complete details.

To run the Linker, give the command

```
.R LINK
*
```

('*' indicates that the Linker is waiting for a command). The first command line can include the output file, a map file if desired, and up to six input files. The file PASCAL.OBJ, which contains the Pascal run-time library, must appear in every Pascal link procedure. The /C switch (continue) allows commands to be continued onto the next line.

```
*OUT,MAP=MAIN,SUB1,LIB1/C
*PASCAL
*^C
```

The overlay facilities of the Linker are selected with the /O:N switch, where the parameter N indicates the overlay region number. Sets of modules that are allocated to the same region will be overlaid against other modules in the same region, with only one set of modules per region actually in memory at any one time.

The following sequence links a main program, an external module, and the Debugger into an overlaid executable file. The main program, external module, and the Pascal library are not overlaid. Debugger modules A and B do not call each other, and are overlaid in region 1. Debugger modules 0-9 do not call each other, and can be overlaid in region 2.

When using overlays in V4 of RT-11, you must use the /T switch to explicitly set the transfer address to the symbol $START. The /T will avoid a bug in the V4 linker.

```
.R LINK
*PROG=MAIN,SUB1,PASCAL/C/T
*P:A/O:1/C
*P:B/O:1/C
*P:0/O:2/C
*P:1/O:2/C
*P:2/O:2/C
*P:3/O:2/C
*P:4/O:2/C
*P:5/O:2/C
*P:6/O:2/C
*P:7/O:2/C
*P:8/O:2/C
*P:9/O:2
Transfer symbol? $START
*^C
```

The Librarian combines relocatable object modules to form object libraries. These libraries may be included as input to the Linker, which will select only those modules needed by the program being linked. Note that a module always consists of the entire set of procedures and functions from its compilation. Individual procedures cannot be selected from a module.

For example, the string package STRING.PAS can be edited to form 9 modules, with each module containing one procedure or function. The 9 modules can then be compiled, and combined into a library as follows.

```
.R LIBR
*STRING=LEN,CLEAR,READS,WRITES,CONC/C
*SEARCH,INSERT,DELETE,SUBS
*^C
```

Embedded Assembly Code

PDP-11 assembly code can be embedded within an Pascal-1 program at any point where a comment might appear. Embedded assembly code takes the form of a special comment beginning with the embedded switch $C, as in the comment

```
{$C MOV %0,-(%6) }
```

The assembly code section extends to the closing comment brace (this closing brace must not be in an assembler comment). Any of the capabilities of the MACRO assembler may be used.

The Pascal-1 compiler scans the embedded assembly code and replaces tokens within the code that correspond to certain classes of Pascal identifiers. This provides simplified access to Pascal data and control structures. However, the programmer is required to have some understanding of the internal structures. See the

section on Run-Time Memory Organization, and examine the code produced by the compiler. Constant identifers appearing in assembly code are replaced by their defined values. Variable identifiers are replaced by the numeric offset from the appropriate base pointer. For global variables, the base pointer is Register 5 (R5); for local variables, the stack pointer (SP) is the base. For example, to swap the halves of a local integer variable I, the code would be

```
{$C SWAB I(SP) }
```

and to assign the constant Ten to the global variable Count one can write

```
{$C MOV #TEN,COUNT(R5) }
```

Any temporary stack usage is not recognized by the compiler, and must be included in indexed addressing of local variables.

Parameters of Pascal procedures and functions are treated as local variables, and are accessible in the same fashion. Internally, a Var parameter is the address of the actual parameter, so references to Var parameters must be indirect, as in

```
{$C MOV @VAR(SP),R0 }
```

Procedure and function identifiers are replaced by the internal label assigned by the compiler. To assign a value to a function, it is best to move the value to a local variable and then use a Pascal assignment statement to copy the value to the function.

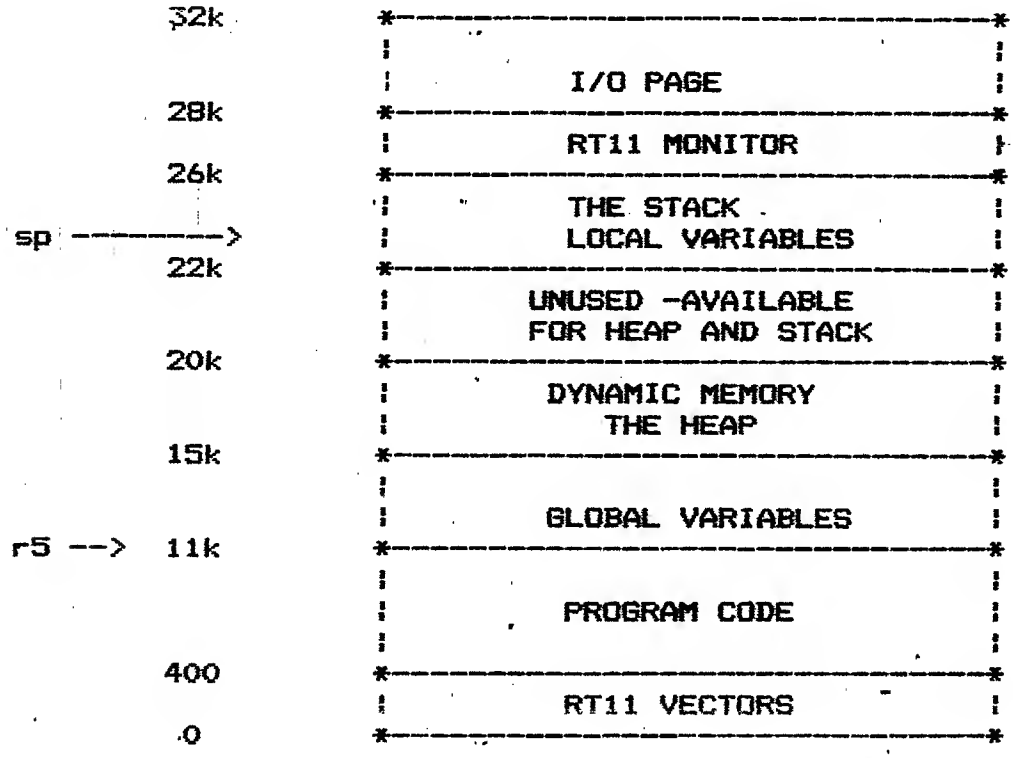
The programmer is responsiblé for selecting the proper base register, as the compiler provides no error-checking capability. Identifier substitution is performed for <u>all</u> identifiers in these classes. This can cause problems if the programmer defines an identifier corresponding to a MACRO instruction, such as a constant named 'MOV'.

With one exception, registers R0-R4 are available within embedded code sections. The exception is With statements, each of which maintains a fixed address in a register. "With" register allocation is in the order R3, R2, R1, R0 and can be determined from the Pascal program. Registers R5 and SP must be preserved across the range of an embedded code section.

The default numeric radix of a Pascal-produced assembly code file is decimal, not the normal octal.

Run-Time Memory Organization

A PDP-11 program has an address space of 32,768 words, or 32KW, or 64KB (1KW is 1024 words). The figure below shows this address space as it might be allocated for a typical program of moderate size.



This figure represents a snapshot taken during program execution, illustrating the partitioning of available memory. Each partition is described in the following sections.

RT-11 Vectors and Communication

The RT-11 Vector partition begins at address zero and occupies the first 256 words of all programs. This area contains interrupt vectors and RT-11 status indicators and also is used for communication between the Pascal program and other programs linked by chaining.

Program Code

The Program Code partition contains all of the instructions of the user program, including overlays, external modules, and routines from the run-time library. The partition is allocated memory adjacent to the RT-11 Vector partition. The size of this

partition depends entirely on the size of the user program.

Global Variables

The Global Variable partition contains all of the program's global variables, those defined in the outermost, or main, block of the program. This partition is fixed in size during program execution.

Register 5 (R5) points to the base of the Global Variable partition and is used for access to global variables.

Dynamic Memory == The Heap

The Dynamic Memory partition contains I/O control blocks and buffers, and variables allocated by the New() procedure. The Heap is unique in that it is not allocated any memory initially, but instead expands as necessary. The Heap is allocated adjacent to the Global Variable partition, and may grow on demand to the upper limit imposed by the Stack partition. The error message 'New() exceeded memory' indicates total exhaustion of memory resources.

Local Variables == The Stack

The Stack partition contains all variables local to inner blocks of the program, and is also used for temporary calculations, parameter passing, and subroutine return information. At the time a block is entered, a stack frame is created. The stack frame contains all information local to that block. Stack frames are created and released in a purely nested fashion. See below for a detailed description of a stack frame.

The current Stack frame is always pointed to by the Stack Pointer (SP, register R6), which initially points to the top of the Stack partition. As nested Stack frames are allocated, the Stack Pointer decreases in value (points to lower addresses). If the Stack partition is too small, the Stack Pointer will eventually overrun the Heap partition and cause the 'Stack exceeded memory' error.

Note that the Stack is located at the high limit of user memory, and that the USR, if swapping, will swap over the Stack. This will cause problems for programs that call the USR via embedded assembly code, or by external non-Pascal subroutines. The simplest solution is to use the 'SET USR NOSWAP' command to make the USR permanently resident. Another solution is, for each USR call, to save the current stack pointer (SP), and then set the SP to low memory (1000) before pushing USR parameters onto the Stack.

RT-11 Resident Monitor

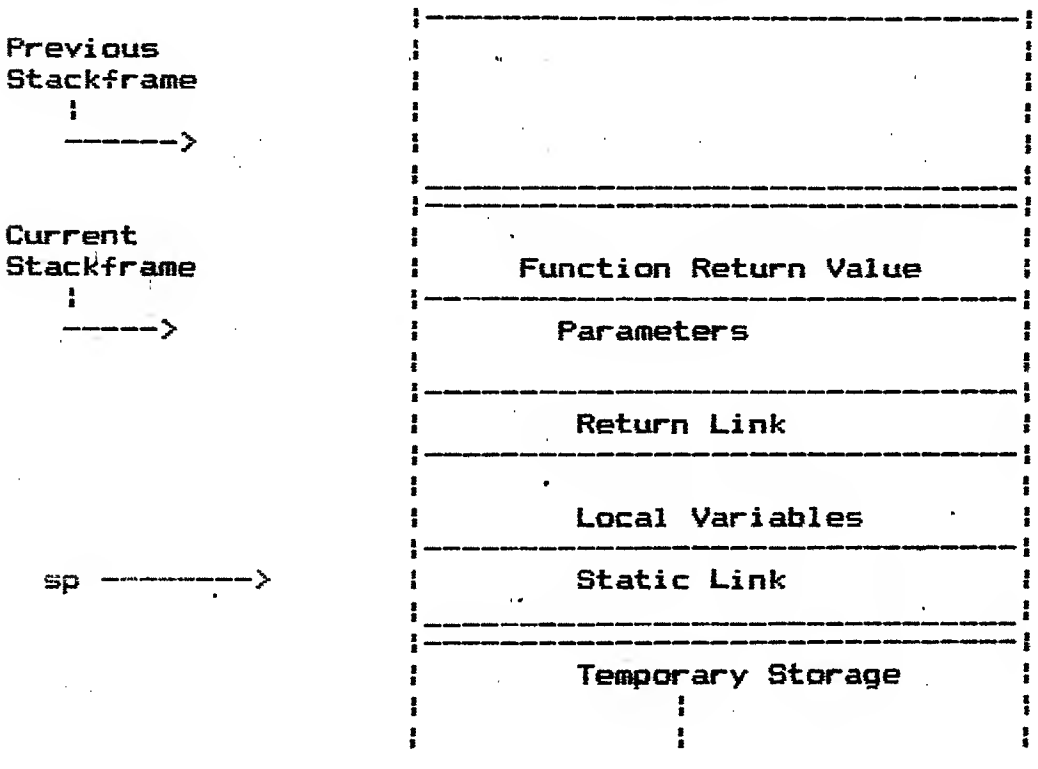
This area is at the high limit of available physical memory, and contains the fixed RT-11 Resident Monitor, the User Service Routine (USR) if it is set NOSWAP, and any device handlers loaded with the LOAD command.

I/O Page

The I/O Page of address space contains all device status and command registers and internal processor registers.

A Closeup of the Stack Partition -- The Stack Frame

The Stack partition is entirely composed of Stack Frames. A Stack Frame is created during entry to every block (excluding the main program block), and is released when the block is exited. The following diagram illustrates the possible components of a Stack Frame. Most of these components are optional. Only the Return Link is required in every Stack Frame.



Function Return Value

This field is present in Stack Frames associated with function blocks, and holds the value to be returned by the function. Its position at the bottom of the Stack Frame allows the field to be 'popped' from the stack when control returns to the caller of this block.

Parameters

The Parameters field contains either parameter values or their addresses. Blocks without parameters do not have this field in their Stack Frame(s).

Return Link

This field is the subroutine return address, where control is transferred on exit from this block.

Local Variables

This field contains all local variables for this block. It does not appear for blocks without local variables.

Static Link

The Static Link appears only in blocks which are lexically enclosed by other procedure or function blocks. The Static Link is used for references to intermediate level variables in the enclosing block(s). The Static Link points to the base of the Stack Frame of the latest invocation of the immediately enclosing procedure or function block, and it is the first link in the Static Link chain.

The Stack Pointer (SP) is also used for transient temporary storage, as in interrupts and Pascal library calls, and each For statement requires 3 words of temporary stack storage during its execution.

Foreground Operation

For foreground operation, you must allocate additional memory to ensure that you will have enough space for Pascal variables and file buffers. Each Pascal file requires about 300 words (more for large buffers), so you should allocate at least 600 words for the default input and output files. Use the /N: switch for V3 and the /BUFFER: switch for V4:

```
.FRUN <file name>/N:1024. (V3)

.FRUN <file name>/BUFFER:1024. (V4)
```

Extended Precision

Values of type Real are normally stored in the PDP-11 single-precision format, which requires 2 words of storage per value and offers 7 decimal digits of precision. The /X compilation switch or the $X embedded switch causes all Real values to have extended precision. Extended precision values each occupy 4 words of storage, and provide 15-digit precision in all real calculations, including the transcendental functions.

Extended precision applies to all Real values in a program. You cannot mix normal and extended precision variables. All external modules must be compiled with the same precision as the main program, even if no Real variables are present.

Compared to normal precision Real variables, extended precision variables require twice the storage. The effect on computation time is dependent on the processor hardware: the FPP floating-point processor provides hardware support for extended precision with a slight performance penalty (30%); processors with the FIS floating-instruction set will experience the most severe relative performance penalty (20 to 1), because FIS offers no extended support and extended calculations are performed entirely in software; processors lacking any floating-point hardware can expect a performance penalty of about 100%. (Even so, the net effect on the program is still very small.)

The System Error() Procedure

When a fatal run-time error occurs, the system procedure Error() is called with parameters describing the error and the system state. The Error() procedure is known by the global name ERROR, and may be replaced by a user-written external module of the same name. The external module must accept the parameters defined below.

```
type Class = (Fatal,IOError,Warning);
     Message = packed array[1..100] of Char;
procedure Error(
    ErrorClass: Class;
    ErrorNumber: Integer;
    ErrorMsgLength: Integer;
var ErrorMsg: Message;
var XFile: Text;
    IOStatus: Integer;
    UserPC: Integer;
    FilenameLength: Integer;
var Filename: Message;
)
```

The ErrorClass parameter indicates the type and severity of the error; Fatal and IOError are errors with no possible recovery, while Warning errors will recover automatically. The ErrorNumber indicates the exact cause of the error. ErrorMsgLength and ErrorMsg define the text of the printed error message normally displayed for this error. The XFile parameter identifies the file variable associated with this error, if any. IOStatus is the value of the RSTS/E I/O status word. UserPC is the program counter saved at this error, which can often be used to identify the program segment responsible for the error. Finally, FilenameLength and Filename describe the external name associated with the file variable XFile.

The possible courses of action available to the Error() procedure are very limited, as exiting from the Error() procedure normally results in termination. The program global variables are available and may aid in diagnosing the problem. The Error() procedure may provide operator interaction or recording capabilities beyond the normal messages to the terminal, and as a final resort may call on operating system facilities to 'chain' and restart the program or initiate another program.

Compiler Error Messages

',' used instead of ';'
8 or 9 in octal constant
Argument must be integer
Argument must be ordinal type
Argument must be real
ARRAY index out of range
ARRAY index type error
Bad ABS argument
Bad argument
Bad CASE label
Bad constant
Bad EXIT
Bad expression
Bad field list
Bad FILE name
Bad FOR statement
Bad FUNCTION name
Bad FUNCTION result type
Bad IN operands
Bad index type
Bad LABEL
Bad ORIGIN for variable
Bad parameter
Bad PROCEDURE name
Bad PROGRAM name
Bad READ statement
Bad RECORD
Bad scalar type
Bad SET element
Bad subrange
Bad TYPE
Bad TYPE specification
Bad variable list
Bad variant
Bad WITH statement
Bad WRITE statement
Boolean expression needed
Constant overflow
Don't repeat FORWARD parameter list
Duplicate CASE label
Duplicate field name
ELSE must be last in CASE
Expression too complex -- out of registers
Expression too complex -- out of registers (real)
Field list must be in parentheses
File variable missing
Format expression must be integer
FORTRAN must be VAR parameters
FUNCTION arg must be real or integer
FUNCTION argument missing

Illegal assignment
Illegal character
Illegal operator
Illegal type of operand
Improper symbol
Incompatible ARRAY type
Incompatible type
Invalid declaration, probably missing END
Invalid symbol
LABEL defined at wrong level
Label must be integer
LABEL not declared
LABEL redefinition
Local VAR definitions must precede PROCEDURE definitions
Missing ')'
Missing ')' at end of list
Missing '.' at end of program
Missing BEGIN
Missing END
Missing END in CASE
Missing field variable
Missing LABEL
Missing label definition
Missing operand
Missing operator
Missing semicolon
Missing UNTIL
Must be simple variable
NEW or DISPOSE arg must be pointer
Not implemented
ODD argument must be integer
Output file error
Source line too long
Strange '[' -- bad SET or missing ARRAY definition
TEXT file expected
Too few arguments
Too many arguments
Too many errors in this line
Too many errors!
Too many levels
Too many symbols
Undefined FORWARD PROCEDURE or FUNCTION
Undefined operand
Undefined pointer base type
Undefined symbol
Unresolved forward type reference
WITH nested too deep

Run-Time Error Messages

Bad filename
Can't Reset(output)
Can't Rewrite(input)
Compiler/library mismatch -- please recompile
Dispose(nil) attempted
Division by zero
Duplicate Dispose()
End of file
Exp() overflow
File not open
Floating-point format error
Get() not allowed
I/O transfer error
Illegal value for integer
Integer overflow
Log() of zero or a negative number
New() exceeded memory
New() of zero length
No channels available
No file to Reset()
Overlay error
Put() not allowed
Real overflow
Reset() failure
Rewrite() failure
Seek() on sequential file
Seek() out of range
Set element out of range
Sqrt() of a negative number
Stack exceeded memory
Subscript out of bounds
Trunc/round overflow
Unexpected trap

OMSI Pascal-1 V1.2 Language Specification

The Language Specification contains details of extensions and limitations of OMSI Pascal-1 as compared to Standard Pascal. Standard Pascal was first defined in the Pascal User Manual and Report by Kathleen Jensen and Niklaus Wirth. A further definition is available in the draft proposed Standard from the British Standards Institution (BSI). The draft BSI Standard is being considered for acceptance as an international standard by the International Standards Organization (ISO) and the American National Standards Institute (ANSI). The original Report and the draft BSI Standard are in general agreement. Where the Report and the Standard differ, this document will give a specific reference.

January 3, 1980

Contents

1.0 Syntax Extensions

This section describes extensions to the formal structure of Pascal which are of general utility.

1.1 Program heading

The program heading is optional in OMSI Pascal-1 programs, and it may be omitted entirely. If the program heading appears, the program name will be printed on each page of the program listing. The first six characters of the name will be used as the external name of the object module. Parameters appearing in the program heading are ignored.

1.2 Declaration ordering

The ordering of global declaration sections (CONST, TYPE, VAR, LABEL) is extended in OMSI Pascal-1. Declaration sections may appear more than once and in any order, so long as identifiers are defined before being used.

One application of this is the concatenation of source modules with main programs which provides a primitive source library capability.

Example - compiler input PLOT,MAIN:

```
(* define source module PLOT *)
VAR  ...   (* global plotter variables *)
PROCEDURE  (* and plotter functions *)
PROCEDURE ...
(* end of plotter module *)

(* program file MAIN *)
VAR  ...   (* global variables *)
BEGIN (* main program code *) END.
```

1.3 Comment brackets

OMSI Pascal-1 provides three forms of comment brackets: the Standard braces {...}, the Standard alternate for upper-case terminals (*...*), and the additional form /*...*/. These may be interchanged freely - it is not necessary for opening and closing comment brackets to have the same form. Comments may not be nested.

Examples:

```
(* This is a valid comment */
{ This is (* not *) a valid comment }
```

1.4 ELSE clause in CASE statements

OMSI Pascal-1 allows an optional ELSE clause to appear in a CASE statement. It indicates a statement which is to be executed if the CASE selector expression does not match the value of any CASE label. If included, the ELSE clause follows all other statements inside the CASE statement. If no ELSE clause appears and no statement is selected, control passes to the statement following the CASE statement.

Example:

```
repeat
  Readln(Ch);
  case Ch of
    'A','a': Append;
    'D','d': Delete;
    'I','i': Insert;
    'N','n': Newfile;
    'Q','q': ;
    else Writeln('"',Ch,'" is not a legal command');
  end;
until (Ch = 'Q') or (Ch = 'q');
```

1.5 EXIT statement

The EXIT statement terminates the immediately enclosing iterative statement (WHILE, REPEAT, FOR).

The EXIT statement is included for compatibility with previous versions of OMSI Pascal-1. Its use is not recommended in programs intended to be portable.

Example (table search):

```
Found := False;
for I := 1 to Tablesize do
  if Table[I]=Key
    then begin
      Found := True;
      exit;
end;
```

1.6 EXTERNAL Procedures and Functions

The keyword EXTERNAL provides access to separately compiled subroutines and to program libraries and overlay facilities. EXTERNAL appears in the place of a procedure or function body to indicate that the procedure or function is compiled separately.

The compiler will generate references to an external (global) symbol. The first six characters of the procedure or function identifier must form a unique external symbol. References to an external procedure or function are resolved at link or task build time.

Note that the compiler is unable to check parameter types at an external interface.

Examples:

```
procedure Erase; external;
function Rad50(A,B,C: char): Unsigned; external;
```

1.7 FORTRAN Procedures and Functions

The directive 'FORTRAN' is similiar to the EXTERNAL directive. The compiler will generate a calling sequence corresponding to the Digital PDP-11 standard calling sequence, with register 5 (R5) pointing to an argument list. The FORTRAN directive enables calling of external MACRO and FORTRAN subroutines. The FORTRAN calling sequence passes parameters by reference, so the corresponding Pascal parameters must be declared as VAR parameters.

The FORTRAN directive generates the proper call sequence for FORTRAN subroutines. It does <u>not</u> provide for initialization of the FORTRAN runtime I/O system.

Example:

```
function Difference(var X,Y: Real): Real; fortran;
```

2.0 Low-Level Interface

The low-level interface section describes those OMSI Pascal-1 extensions which are useful to programmers who need access to machine dependent PDP-11 characteristics.

2.1 Octal (Base 8) Numbers

Integer constants may be written in octal notation by appending the capital letter 'B' to the number. This applies only to compile-time constants -- runtime integer conversions via Read() are performed using decimal notation.

```
Example:  const TabCode = 11B;  (* ASCII tab character *)
```

2.2 Unsigned Integers

The predefined type Integer has the subrange (-32768 .. 32767) and uses the PDP-11 signed arithmetic operations. Unsigned integers may be specified with the subrange 0..65535. The compiler will generate the unsigned comparison operations of the PDP-11 and will not detect multiplication and division overflow of unsigned integers.

Unsigned integer operations apply only to integer calculations. I/O conversions and conversions to and from Real values are always signed integer operations.

```
Example:  type Unsigned=0..65535;
```

2.3 Logical operations on Integers

The Boolean operators AND, OR, and NOT are extended to Integer operands. The operators perform the Boolean operations on all 16 bits of their operands. This allows testing or setting of individual bits within a word (for instance, status bits within a device register).

```
Example:  Byte := Ord(Ch) and 377B;
```

2.4 References to fixed (absolute) memory

OMSI Pascal-1 allows the keyword ORIGIN to appear in variable declarations, associating a variable identifier with a specific memory address. This provides access to fixed memory addresses,

such as device control registers or operating system parameter blocks.

Example (read directly from the RT11 console):

```
const Ready=200B;
var   KbCsr origin 177560B, KbBuff origin 177562B: Integer;
      Ch: Char;
begin
      while (KbCsr and Ready)=0 do (* nothing *);
      Ch := Chr(KbBuff);           (* get character *)
end;
```

2.5 Address operator (@)

OMSI Pascal-1 provides a unary address operator, indicated by the @ character. When applied to a variable of type T, it yields a value of type ^T (pointer to T). The address operator can be used to link variables into list structures or (more commonly) to pass variable addresses to low-level routines.

Example:

```
var  Buffer: Block; XRLoc origin 446B: ^Block;
begin
    XRLoc:= @Buffer;  (* pass address to RSTS/E *)
end
```

2.6 Embedded assembly code

PDP-11 MACRO assembly code may appear at any point in an OMSI Pascal-1 program. Assembly code sections have the form of a Pascal comment, beginning with the $C embedded switch. Any MACRO-11 feature may be used within embedded code. The compiler provides some assistance in accessing Pascal variables, though the programmer is expected to have some understanding of the OMSI Pascal-1 runtime environment. Note that the default radix within a Pascal-produced MACRO file is decimal, not octal.

Example:

```
procedure EmtTrap(N:Integer);
begin
  (*$C
    MOV N(SP),-(SP) ; push parameter N
    EMT 53          ; call EMT handler
  *)
end (*EmtTrap*);
```

3.0 I/O Support Extensions

I/O support extensions provide the OMSI Pascal-1 programmer with additional control of the interface to the operating system.

3.1 Reset()/Rewrite() optional parameters

Three additional parameters may appear following the file variable in calls to the Reset() and Rewrite() standard procedures. These optional parameters allow the program to dynamically bind a file variable to an external file and provide status and error information.

The general form is:

```
Reset( F , Filename , DefaultName , Size )
```

where the parameters have these types:

```
F - any file variable
Filename - literal string, or (packed) array of Char
DefaultName - same as Filename
Size - Integer variable
```

Reset(F,Filename) connects the file variable F with the external file identified by Filename. Filename conforms to the operating system conventions, and may contain device, filename, extension, and other fields such as PPN/UIC and version number. The Filename parameter may also contain switches specifying access modes or other special characteristics. If the external file does not exist prior to the Reset(), a fatal error will result. Upon sucessful completion of a Reset(), either the file buffer F^ will contain the first element of the file, or Eof(F) will be True.

Reset(F,Filename,DefaultName) performs the same function, with DefaultName having the same format as Filename. Fields of the external name which are not specified in Filename are filled from the information in DefaultName. Common default fields are the extension, protection code, and mode switches.

Reset(F,Filename,DefaultName,Size) provides a recovery capability on file open errors. Size must be a variable (VAR parameter). After a succcessful Reset(), Size contains the length of the file in blocks. If an error occurs, Size is set to negative one (-1).

```
Rewrite( F , Filename , DefaultName , Size )
```

Rewrite() creates a new external file. The optional parameters have the same meaning as in Reset() with one addition: Size specifies the initial storage, in blocks, to be allocated for the file.

Reset() and Rewrite() may be applied to the standard files Input and Output respectively. This will redirect the default input or output streams to the specified file instead of the user terminal. A subsequent Close() will break the connection and reconnect the default file to the terminal.

Example:

```
program Copy; (* copy to printer *)
var Name: array[1..20] of Char;
    Ch: Char; Len: Integer;
begin
  repeat                        (* Get a Filename and Reset() it *)
    Write('File: ');
    Readln(Name);
    Reset(Input,Name,'.PAS',len)
  until Len <> -1;              (* until not error code *)
  Rewrite(Output,'LP:');        (* redirect output to printer *)

  while not Eof do begin        (* copy Input to Output *)
    while not Eoln do begin
      Read(Ch); Write(Ch);
    end;
    Readln; Writeln;
  end;
end.
```

3.2 Seek() procedure

The predefined procedure Seek() causes direct positioning of a file window variable to any desired component of the file.

```
Seek( F , Index )
```

F may be of any file type except Text, and must be connected to an external file which supports direct access (typically disk or DECtape). Index is an unsigned integer expression which specifies the desired component. File components are numbered sequentially beginning with one (1). If Index specifies a number greater than the number of components actually present, then Eof(F) is set to True.

To read component N of file F, use:

```
Seek(F,N); (* component N is available in F^ *)
```

To write component N, use the sequence:

```
Seek(F,N); (* position to component N *)
F^ := ();  (* assign new value *)
Put(F);    (* write component to file *)
```

If the Put() in the above sequence is omitted, the effects will be unpredictable and the new data may be lost.

Sequential I/O operations such as Get() and Put() may be mixed with Seek() and will advance the file window to the next component. Reset(F) is equivalent to Seek(F,1).

The direct access extension bypasses the Standard Pascal restriction prohibiting simultaneous read and write access to a file. For this reason, direct access files are identified by the '/Seek' switch which must appear in the Filename or DefaultName field of the associated Reset() or Rewrite().

3.3 Break() procedure

For efficiency, OMSI Pascal-1 buffers transmitted data. Break(F) forces the actual transmission of data from a partially filled buffer of file F. This can be useful with interactive terminals, or to guarantee actual transmission of data to a shared disk file.

3.4 Close() procedure

Close(F) indicates that the program has completed processing the file F, and that internal buffer storage may be reclaimed. Close(F) removes any connection to an external file, so that Reset(F) or Rewrite(F) must precede any subsequent operations with that file variable.

3.5 Readln() Array of Char

Read() and Readln() will read characters from a Text file into a (packed) array of characters. Reading begins at the current file position and continues until either the array is filled, or Eoln() is True, in which case the remainder of the array is filled with blanks.

3.6 Write() Array of Char

In accordance with the draft proposed ISO Standard, a Write() procedure call applied to an array of Char will truncate the written string if the field width parameter will not allow the entire string to be written.

Example:

```
Write(Buffer:BuffCount);  (* write buffered characters *)
```

3.7 Write() Octal (Base 8)

Write() will write integers in octal notation if the field width specification is negative.

Example: `Write(I:-5);  (* Display octal value of I *)`

3.8 Interactive I/O

The Pascal Standard requires that the first element of a file be available as soon as the file is Reset() (the buffer variable F^ is assigned a value immediately). This can present serious difficulties when applied to files which are interactive terminals. For example, if the default input file is the user's terminal, the standard can be interpreted to require that the user type the first input character (or line) prior to the execution of the first program statement.

OMSI Pascal-1 takes the following route around the problem. When an interactive file is Reset(), the buffer variable is set to a space and Eoln(F) is set to False, but no actual I/O transmission occurs. Each Read() request then waits for sufficient data to satisfy the request, but no more.

This solves most of the problems with interactive terminals in a predictable manner, but one should note that this approach creates other difficulties. When applied to an interactive file, the following program is unable to distinguish between an empty line and a line containing a single space. This is because Eoln() cannot be set until the end of line character is typed to satisfy the Read() request.

Example: (the standard schema for reading a line of characters)

```
var Line: array[1..72] of Char;
    Count: Integer;
begin
  Count := 0;
  while not Eoln do begin
    Count := Count+1;
    Read(Line[Count]);
  end;
  Readln;
end;
```

4.0 Additional Predefined Functions

OMSI Pascal-1 provides some additional built-in functions.

4.1 Time function

The Time function takes no parameters and returns a real value which corresponds to the current time of day. The Time is represented in hours after midnight, so that 9:30 AM is 9.50 and 1:45 PM is 13.75. The exact resolution of the Time function is dependent on the operating system, but all operating systems provide a resolution of at least one second.

Example:

```
procedure WriteTime;
var Hrs, Mins: Integer;
    AmPm: array[1..2] of Char;
begin
  Mins := Round(Time*60);
  Hrs := Mins div 60;
  Mins := Mins mod 60;
  if (Hrs < 12)
    then AmPm := 'AM'
  else if (Hrs = 12) and (Mins = 0)
    then AmPm := 'M '
  else AmPm := 'PM';
  Write('At the tone the time will be: ');
  Write(((Hrs+11) mod 12 + 1):2);
  Write(':', Mins div 10:1, Mins mod 10:1, AmPm:3);
  Writeln(Chr(7));
end;
```

"At the tone the time will be: 11:56 AM"

4.2 Exp10() and Log() functions

The Exp10() and Log() functions are similiar to the standard Exp() and Ln() functions, but with a logarithm base of ten (10).

5.0 Non-Standard Language Elements

This section describes the elements of OMSI Pascal-1 which do not conform to the accepted definition of Standard Pascal.

5.1 Pack() and Unpack() not available

The reserved word PACKED may appear in type definitions, but it has no meaning in OMSI Pascal-1 programs. Packed types require the same amount of storage as unpacked types. The standard procedures Pack() and Unpack() are not available. The following equivalent FOR statements can be used instead:

```
var A: array[M..N] of T;
    Z: packed array[P..Q] of T;
  for J:= P to Q do Z[J]:= A[J-P+I]; { Pack(A,I,Z) }
  for J:= P to Q do A[J-P+I]:= Z[J]; { Unpack(Z,A,I) }
```

5.2 Program Parameters

Program parameters (identifiers appearing in the program heading) have no meaning in OMSI Pascal-1 programs. The program heading may be omitted entirely if desired. External files can be declared by using the Reset() and Rewrite() procedures with optional parameters.

5.3 Identifier Scope Rules

In Standard Pascal, the scope of an identifier (that section of the program within which the identifier indicates a particular object) is directly related to the block structure. A definition of an identifier in a procedure, for example, prohibits that identifier from indicating another object throughout the entire procedure.

OMSI Pascal-1 uses a subtly different rule for the scope of an identifier, called 'one-pass' scope, in which a definition of an identifier prohibits only subsequent uses of the identifier within the block from indicating an object outside the block.

The non-standard scope rule is described here for completeness, but it is of little concern to the programmer. Indeed, the majority of Pascal compilers use the identical (incorrect) rule.

5.4 Read()/Write() Text files only

In the 1978 printing of the Pascal User Manual and Report, the Read() and Write() standard procedures were extended to apply to all file types. This extension has not yet been incorporated into OMSI Pascal-1, so that Read() and Write() are applicable only to files of the standard type Text.

The following substitions may be used:

For Read(F,V), use: V:=F^; Get(F);

For Write(F,V), use: F^:=V; Put(F);

5.5 Eof() not accurate (RT11, RSTS only)

On the RT11 and RSTS operating systems, a file is structured as a sequence of 512 byte blocks. No finer resolution is available as to the end of data in the last block. Therefore, the Eof() standard function can not be relied upon as accurate, and another method (sentinel record, record count) should be used to indicate the end of usable data.

Note that this problem does not apply to Text files, where Eof() is identified correctly.

6.0 Implementation Definitions

This section provides specific details and characteristics of implementation-defined elements of OMSI Pascal-1.

6.1 Identifiers

OMSI Pascal-1 permits identifiers to be of any length, and all characters are significant. Lower case letters may be used and are interpreted the same as upper case, so that "name", "Name", and "NAME" are equivalent identifiers.

Due to limitations of the object program file structures, the first six characters of any EXTERNAL or FORTRAN identifier must form a unique external name.

6.2 Standard type Integer

The standard type Integer has the range (-32768..32767). Unsigned integers may be declared using the subrange notation 0..65535. Note that arithmetic overflow is detected only for multiplication and division of signed integers.

The predefined identifier Maxint has the value 32767.

6.3 Standard type Real

Real variables have the standard PDP-11 single or double precision floating point structure, with the range 1E-38 .. 1E+38. Single precision values give 7 decimal digit precision; extended (double precision) values give 15 digit precision. Arithmetic overflow is detected for all real operations, but underflow is ignored and gives a result of zero.

The standard transcendental routines are accurate to 6 decimal digits in single precision, and 15 decimal digits in extended precision.

6.4 Standard type Char

OMSI Pascal-1 uses the 7-bit full ASCII character set. Characters are stored as signed bytes with all 8 bits available to the programmer, so that Ord(Char) has the subrange (-128..127).

6.5 Standard type Text

The standard type Text is a file type with components of type Char, with the characters masked to the 7-bit ASCII set, and skipping the null (0) character. On RSX systems, the standard function Eoln() is set by the end of a file record; on RSTS/E and RT11 systems by the LF (10) or ESC (27) character codes.

The standard procedures Read(), Readln(), Write(), Writeln(), and the standard function Eoln() are applicable only to Text files. The Seek() procedure is not recommended for use with Text files.

6.6 SET types

OMSI Pascal-1 limits sets to a maximum of 64 elements. The 64 element maximum forms a subrange which is not required to have a lower bound of zero, but may instead be positioned at any 64 element (or smaller) subrange of a base type (for example: 100..150, -25..25).

A set of the standard type Char is equivalent to the set of Chr(32)..Chr(95), which is a subset of ASCII containing the upper case letters, digits, punctuation symbols, and the space character, but lacking the control characters and lower case letters.

6.7 New() and Dispose() procedures

In allocating storage for variant records, the New() procedure will allocate memory for the largest variant; any tag field values specified to New() and Dispose() are ignored.

Storage must be explicitly released with Dispose() -- no automatic garbage collection is performed. Storage occupied by variables passed to Dispose() is reclaimed for use by the New() procedure. Dangling pointer references are not detected.

6.8 Procedural Parameters

The passing of PROCEDURE and FUNCTION parameters is supported by OMSI Pascal-1 with the syntax described in the Pascal User Manual and Report (the proposed ISO Standard differs in this area).

Predefined procedures and functions are not permitted as procedural parameters. This can be bypassed by declaring a second procedure which calls the standard procedure, and which can itself be used as a procedural parameter.

Example:

```
function Sine(X: Real): Real;
begin
  Sine:= Sin(X)
end;
```

6.9 Implementation Limitations

The PDP-11 has six general purpose registers. In OMSI Pascal-1, one register (R5) is always allocated for access to global variables, and another (R4) is allocated in some blocks for access to intermediate level variables. The remaining registers are used for integer calculations, address computations, and WITH statement variable access. Each WITH statement uses one register for the duration of the enclosed statement. This implies a maximum nesting of WITH statments, of three levels. Complex expression calculations can also exceed the available registers. If the 'Out of registers' error occurs, remove a WITH statement or simplify the indicated expression by calculating intermediate results.

The syntactic nesting of procedures is limited to a depth of 10 levels. There is no implementation restriction on the actual depth of recursion of a program, although unlimited recursion will eventually cause the program to exceed available memory.

6.10 Error Detection

OMSI Pascal-1 does not detect the following runtime errors:

Uninitialized variables
Subrange bounds exceeded
Integer overflow
Real underflow
Record variant mismatch
Dereference of NIL pointer

The following runtime errors are detected:

Stack overflow
Heap overflow
Real overflow
Integer multiply/divide overflow
Array bounds exceeded
Dispose() of NIL or duplicate pointer
Incorrect numeric format
I/O errors

Table A: Predefined Identifiers

Predefined Identifiers

Constants	
False,True	
Maxint	
Types	
Boolean	
Char	
Integer	
Real	
Text	
Variables	
Input, Output	
Functions	
Abs	
Arctan	
Chr	
Cos	
Eof	
Eoln	
Exp	
Exp10	Base 10 Exponential
Ln	
Log	Base 10 Logarithm
Odd	
Ord	
Pred	
Round	
Sin	
Sqr	
Sqrt	
Succ	
Trunc	
Time	Time of day
Procedures	
Break	Transmit buffered output
Close	Close file
Dispose	
Get	
New	
Page	
Put	
Reset	
Rewrite	
Read	
Readln	
Seek	Direct access I/O
Write	
Writeln	

Reserved Words
(* extensions)

And
Array
Begin
Case
Const
Div
Do
Downto
Else
End
Exit *
External *
File
For
Fortran *
Forward
Function
Goto
If
In
Label
Mod
Nil
Not
Of
Or
Origin *
Packed
Procedure
Program
Record
Repeat
Set
Then
To
Type
Until
Var
While
With

SECTION FOUR: PASCAL ON-LINE DEBUGGER (POD)

Introduction to the Debugger

The Pascal On-Line Debugger (POD) is a symbolic debugging tool that lets you interactively control the execution of your Pascal program. You can suspend execution at particular statements, execute one statement at a time, and examine and modify the values of particular variables. Since POD traps errors and identifies the last statement executed, you can easily pinpoint the source of run-time errors.

POD is really a series of Pascal procedures that are linked with a program. When you specify the debugging option (/D), the Pascal compiler includes a call to POD before each procedure and statement in your program. This lets POD control program execution. The compiler also produces a symbol table file containing the definitions and locations of all variables and procedures in your program. Using this, POD can find and modify variables and refer to procedures by name.

The Debugger Guide assumes that you are familiar with the User's Guide and the Programmer's Guide in this manual.

In examples, underlining is used to show the text that you should type. Non-underlined text shows the prompts or other responses by the computer.

Including POD in Your Program

To use POD, you must compile your program with the debugging switch, /D. This automatically generates a symbol table file for your program. For example:

```
.R PCL
*TRIM/D
```

The /D switch causes debugging instructions to be included in the compiled program. The /D switch also produces a debugging file (TRIM.SYM) containing the symbol table information for the procedures and variables of TRIM.

POD supports an option called source debugging, selected with the /S compilation switch. This lets POD print the Pascal source lines associated with the compiled statements in your program. With the /S switch you can debug a program without having to print a listing of the program. The cost for using source debugging is an increase in the size of the program being debugged and a somewhat slower execution speed. All of the examples in this section use the source debugging option.

If you wish to use the source debugging option, specify both the /S and /D switches in the compilation command:

```
.R PCL
*TRIM/S/D
```

When the /S source debugging option is selected, a listing file (TRIM.LST) is automatically created. POD reads this file to display the source program for each Pascal statement. If the listing file is deleted, source debugging is automatically disabled, and POD will then identify statements only by procedure name and statement number.

Running POD

When your program starts, POD will identify itself and ask you for the name of your program. It is assumed that the symbol file and listing file (if the S option is invoked) will share the program name. If either file cannot be found, POD will ask specifically for the necessary file name. If POD asks for a listing file and none exists, give a carriage return. This will cancel the source debugging option. POD will then ask for a symbol file name. Here is a typical POD opening dialogue:

```
RUN TRIM
POD (Pascal On-line Debugger) - 24-Apr-79
POD - program name? TRIM
}
```

When POD is ready to accept commands, it will prompt you with a right brace (}). On some terminals this will print as a right square bracket (]). Commands to POD may be typed in either lower or upper case, and spaces in the commands are ignored. Several POD commands can be typed on the same line if you separate each command with a semicolon (;).

POD commands are presented alphabetically beginning on Page 77.

Accessing Pascal Statements

POD identifies Pascal statements by the name of the procedure containing the statement and the number of the statement in the procedure. The statement number can be found in the column labeled STMT in the listing file produced by the Pascal compiler. Statements in the main body of a Pascal program are considered to be in the procedure MAIN. All Pascal programs begin executing at MAIN,1.

If the source debugging option is being used, POD will print the source line along with the procedure name and statement number.

Pascal allows you to define procedures that define other local procedures. In this way you can create a program containing several procedures all having the same name. However, we strongly recommend that all of the procedures in your program have unique names in order to avoid confusion during debugging.

Accessing Pascal Variables

POD lets you access the variables in your program in much the same way as you use variables in Pascal. Variables and procedure parameters are identified by name, such as MARGIN, LIMIT, or SHOESIZE. Records are specified with the standard dot notation, such as: COORD.X, and RANGE.TOLERANCE.LOW. POD will generate an error message if too few (or too many) fields are specified for a record. Arrays of multiple dimensions are allowed, and POD will check the data type and limits of each index when accessing arrays. Pointers are specified in the usual way. The value of the pointer itself is interpreted as a decimal integer. A nil pointer has a value of zero, and POD will generate an error message if a reference through a nil pointer is attempted.

You can access complex structures by combining several of the structures described above. In general, POD can access a variable in a structure in the same way as that variable is used in your program. Examples of legal variables are shown below:

```
FEET
A.B.C.D
CHIP^.TEMPLATE[3,1,-5].FLUX
PTR^.SON^.SON^.SON
```

Integers are treated as 16-bit signed numbers. The letter "B" placed after an integer (377B, for example) indicates an octal value. Boolean variables take values of either TRUE or FALSE. Character data, including character strings, are always enclosed within single quotes as with 'X' and 'THIS IS A TEST'. Spaces are not ignored within a character string. Real variables are used in the usual way. POD also can access scalar types defined by the user. For example, consider the program section below:

```
TYPE
    COLOR=(RED, WHITE, BLUE);
VAR
    X: COLOR;
```

When POD displays the value of X, it will correctly print the scalar type of X. This capability is provided only by POD — Standard Pascal does not permit output of scalar types.

```
} X:=RED
} W(X)
RED
}
```

POD has another facility not available to the Pascal programmer: its ability to display the value of sets. The ".." notation for included set elements is available for both the input and output of set values.

```
TYPE
    COLOR=(RED, ORANGE, YELLOW, GREEN, BLUE);

VAR
    RB: SET OF COLOR;
    VALUES: SET OF INTEGER;
    Q: SET OF CHAR;
```

These variables may be accessed by POD as shown below:

```
} RB:=[RED..YELLOW,BLUE]
} W(RB)
[RED..YELLOW,BLUE]
} VALUES:=[1..20,50,40,30]; W(VALUES)
[1..20,30,40,50]
} Q:=['E','A','C','F','B','D']
} W(Q)
['A'..'F']
}
```

As demonstrated above, POD lets you assign values to variables in the same way as you assign values to variables in your program. The only restriction is that you cannot evaluate expressions such as C:=A+B, and you cannot call functions such as R:=SIN(3.1415).

POD enforces the Pascal scope rules. In general, this means that at any point in your program you can only access the variables that the program itself can access at that point. Global level variables, those defined at the start of the program, are always available. However, as different procedures are executed, the local variables and arguments of those procedures are temporarily available, while the local variables in procedures not being executed are never available. If you try to use a variable that is not available, POD will print a "symbol not found" error message. Remember, at any statement, you can only use the variables that are available to the program at that point.

POD lets you directly address memory locations as integers. For example, 1234B:=240B modifies location 1234 (octal) to contain 240 (octal). This feature is most commonly used when you deal with pointers. However, be careful, for you might accidentally modify a location within your program and cause unpredictable results.

B(): Set/Clear Breakpoints

The "B" command sets a breakpoint at a particular statement within a program. Before executing each statement in your program, POD checks to see whether a breakpoint has been set at that statement. If a breakpoint has been set, POD suspends the execution of the program and enters command mode. At this point you can examine and alter variables, check the history of the program's execution, or continue the execution of the program.

To set a breakpoint at a statement, type a "B" followed by the statement identifier (procedure and statement number) contained within parentheses. POD will interrupt the execution of your program just before the statement at which a breakpoint is set. Up to eight breakpoints may be in effect at any one time. Examples:

```
} B(MAIN,1)
} G
Breakpoint at MAIN,1  BEGIN I:=0;
} B(INIT,5); C
Breakpoint at INIT,5  PARAM1:=0; PARAM2:=0;
}
```

(The examples above show how the source debugging option works. When POD stops at breakpoint, it prints the Pascal source line for that statement.)

The "G" command in the example starts program execution. The "C" command continues from the breakpoint.

POD has the capability to execute a series of POD commands when a breakpoint is encountered. This facility, called stored commands, is specified when you place the command within angle brackets (< >) after the break command as shown here:

```
} B(MAIN,6) < W(DEPTH); DEPTH:=5 >
} B(POSITION,32)<W(X,Y);C>
```

The first example displays the value of the variable DEPTH then assigns the value of 5 to DEPTH each time the program comes to the statement at MAIN,6. The second example displays the values of the variables X and Y and then continues the execution of the program. In this case POD will not stop and enter command mode. Instead, each time the program comes to the statement at POSITION,32, the variables X and Y will be displayed and the program will continue.

Any POD command may appear in a stored command, but stored

commands may not be nested; i.e., a stored command may not define other stored commands. As many POD commands as will fit on a single line may be specified in a stored command.

There are two ways to cancel a breakpoint. The "K" command described below can be used to kill all breakpoints or just a single breakpoint. However, if the program has just been interrupted because a breakpoint was reached, that breakpoint can be canceled by means of the "B" command with no arguments.

```
} B(MAIN,1)
} G
Breakpoint at MAIN,1  BEGIN I:=0;
} B
} C
```

The "D" command may be used to display the currently active breakpoints and their associated stored commands.

C: Continue Execution

If the execution of your program has been suspended by POD, you may use the "C" command to resume execution of the program. If your program has not started executing, either the "C" or the "G" command may be used to start the program. The section above describing breakpoints has several examples that use the "C" command. Once your program has terminated and POD has re-entered command mode, any attempts to continue the program with the "C" command will be ignored. (There is nowhere to go!) The program may, however, be restarted with the "G" command described below.

If you set a breakpoint inside a loop, it is sometimes desirable to let the statement at the breakpoint execute several times before stopping. One way to do this is to use the "C" command several times to continue from the breakpoint until the desired iteration in the loop is reached. Another solution is to use a repeat count contained inside parentheses after the "C". The repeat count tells how many times the statement at which the breakpoint has been set should be executed before the breakpoint takes effect. For example, you can set a breakpoint at COUNT,10 inside a loop structure. When the loop is first entered, POD will stop the program at COUNT,10 with a breakpoint. The command C(6) will let the loop iterate 6 times before the program stops again at COUNT,10 with a breakpoint. Each of the eight breakpoints has its own repeat count.

D: Display POD Parameters

The "D" command displays the watched variables, labels, and breakpoints that are currently active. Watched variables are described below in the section about the "V" command. Labels are discussed below in the sections about the "G" and "L" commands. The stored commands associated with breakpoints and the watched variable are also displayed.

```
} D

Watching: B[5] <W(B[6],B[7],B[8])>

Breakpoints:
MAIN,13 <W(FOO);C>
MAIN,20
ERR,5  <W(ERRORCODE);H>

User-defined labels:
1:  MAIN,1  BEGIN I:=0;
5:  RETRY,3  RESET(F,NAME,'DAT',STATUS);
}
```

G, G(): Go or Go to a Label

The "G" command without arguments starts or restarts your program at MAIN,1. If the "G" command is followed by a label number in parentheses, the program will be continued at that user-defined label. Do not confuse user-defined labels with Pascal statement labels. User-defined labels are created with the "L" command dynamically as POD controls your program. Pascal statement labels are defined in your source code and are used by the Pascal compiler to generate targets for the Pascal GOTO statement. POD does not use Pascal statement labels.

The "L" command labels the program statement about to be executed. The most common way to define a label at a particular statement is to set a breakpoint at that statement, execute the program until that statement is reached, and then use the "L" command to define the label.

The "G" command should be used with care. It is not always possible to branch from any Pascal statement to any other Pascal statement. Labels follow the same scope rules as variables, so depending on which procedures are being executed, some labels may not be available. If you try to go to a label that is not available, POD will respond with the error message "You can't get there from here". One reason that POD cannot go to a particular label is that if the label is in a procedure that is not being executed, POD is not able to invent the values of the local variables associated with that procedure.

```
} B(MAIN,5); C
Breakpoint at MAIN,5  J:=SIN(Q);
} L(3); B(MAIN,27); C
Breakpoint at MAIN,27  WRITELN('X>Y');
} G(3)
Breakpoint at MAIN,27  WRITELN('X>Y');
} G
Breakpoint at MAIN,5  J:=SIN(Q);
}
```

H: Print Program Execution History

POD maintains a list of the last 10 statements executed by a program. This history is useful in determining how the program reached a breakpoint or how it reached a statement that caused an error. The "H" command prints the history and also the procedure execution stack. The stack shows the procedure and function nesting all the way back to the main body of the program.

```
} B(EVALUATEBOARD,1);C
Breakpoint at EVALUATEBOARD,1  FOR I:=-5 TO 49 DO BMAN[I]:=FALSE;
} H
Program execution history

GENMOVE,3  BEGIN
GENMOVE,4  FATHER:=F;
GENMOVE,5  MOVE:=I*256+J;
GENMOVE,6  OLDPIECE:=B[I]; B[I]:=EMPTY;
GENMOVE,7  OLDPIECE:=B[I]; B[I]:=EMPTY;
GENMOVE,8  IF TURN=BLACK THEN
GENMOVE,9  IF J<=8 THEN B[J]:=BLACKKING ELSE B[J]:=OLDPIECE
GENMOVE,11  IF J<=8 THEN B[J]:=BLACKKING ELSE B[J]:=OLDPIECE
GENMOVE,15  VALUE:=EVALUATEBOARD(ENEMY);
EVALUATEBOARD,1  FOR I:=-5 TO 49 DO BMAN[I]:=FALSE;

Procedure execution stack

EVALUATEBOARD,1  FOR I:=-5 TO 49 DO BMAN[I]:=FALSE;
GENMOVE,15  VALUE:=EVALUATEBOARD(ENEMY);
MOVEPIECE,11  IF MOVESALLOWED THEN GENMOVE(I,J);
EXPAND,15  IF COLOR[WHO]=TURN THEN MOVEPIECE(I,I,0,0);
MAIN,7  EXPAND(ROOT,TRUE);
}
```

K, K(): Kill Breakpoints and Labels

When the "K" command is given without arguments, all label definitions and breakpoints are deleted. When the "K" command is followed by a statement identifier, the breakpoint at that statement is removed.

```
} B(MAIN,5)
} K(MAIN,5)
} B(MAIN,17)
} K
```

Individual breakpoints also can be removed with the "B" command.

L(): Label a Statement

You may label up to eight statements with the "L" command. Labels are used as targets of the "G" command. The label number (1 through 8) is placed in parentheses after the "L". The "L" command always defines the label at the current location within the program being executed. Check the description of the "G" command above for a warning about branching within a Pascal program. The "D" command may be used to list the currently active labels.

```
} B(MAIN,13); G
Breakpoint at MAIN,13  A:=1;
} L(1)
} B(MAIN,15); C
Breakpoint at MAIN,15  B:=37;
} L(5)
} D

Breakpoints:
MAIN,13
MAIN,15

User-defined labels:
1:  MAIN,13  A:=1;
5:  MAIN,15  B:=37;
}
```

P, P(): Execute One Statement in Current Procedure

The "P" command executes a single statement in the current procedure. "P" will not single step through functions and procedures nested in the current procedure, but instead will treat their calls as single statements. If the current procedure ends, "P" will begin single-stepping the procedure that called the current procedure. (Compare "P" to the similar "S" command described below.)

If a repeat count is given in parentheses after the "P", the specified number of statements will be executed before stopping. As with the "C" command, you may not proceed past the end of the program once the program has terminated. Use the "G" command to restart the program.

```
} P
Breakpoint at MAIN,1  BEGIN I:=0;
} P
Breakpoint at MAIN,2  J:=RANDOMINTEGER(3);
} P
Breakpoint at MAIN,3  K:=J*J-I;
} P(5)
Breakpoint at MAIN,8  IF K<J THEN BEGIN
}
```

R: Register Dump

The "R" command prints the values of the processor registers R0-PC in both octal and decimal. This command is normally useful only to those programmers who include in-line assembly language code in their Pascal programs.

S, S(): Single Step

The "S" command is identical to the "P" command above, except that if a statement being stepped through calls another procedure or function, then the new procedure or function also will be executed one step at a time. As with "P", a repeat count may be specified.

```
} S
Breakpoint at MAIN,1  BEGIN I:=0;
} S
Breakpoint at MAIN,2  RANDOMINTEGER(3);
} S(1)
Breakpoint at RANDOMINTEGER,1  BEGIN RANDOM:=X;
}
```

T(): Trace Mode

"T(TRUE)" turns on statement trace mode, while "T(FALSE)" turns it off. When trace mode is on, POD will print the location of each statement before it is executed. If several Pascal statements appear on the same line in the source file, and if those statements are each executed in sequence, then the line containing those statements will be printed only once.

```
} B(MAIN,6)
} T(TRUE)
} G
MAIN,1  BEGIN I:=0;
MAIN,2  J:=0; K:=0; L:=3.14159;
MAIN,5  WRITELN('HI THERE');
HI THERE
Breakpoint at MAIN,6  WRITELN;
}
```

V(): Variable Watch

The "V" command makes POD watch the value of a variable. Before each statement in your program is executed, POD compares the current value of the variable with the value it had when the "V" command was given. If the value has changed, POD stops your program and tells you so. If you continue your program, POD will continue watching for a change in the variable.

The "V" command is useful if your program is malfunctioning because the value of some critical variable is being destroyed somewhere. The "V" command also can be used to watch locations in low memory to detect the incorrect use of a nil pointer.

```
} V(DEPTH)
} C
Value of "DEPTH" changed at statement:
DESCEND,1  DEPTH:=DEPTH+1;
Old value: 0
New value: 1
Breakpoint at DESCEND,2  IF DEPTH>MAXDEPTH THEN
} C
Value of "DEPTH" changed at statement:
DESCEND,1  DEPTH:=DEPTH+1;
Old value: 1
New value: 2
Breakpoint at DESCEND,2  IF DEPTH>MAXDEPTH THEN
} C
Value of "DEPTH" changed at statement:
DESCEND,38  DEPTH:=DEPTH-1;
Old value: 2
New value: 1
Breakpoint at DESCEND,39  END;
}
```

Stored commands may be specified with the "V" command in the same way as with the "B" command. The "D" command will list the name of the variable being watched and the stored commands if any were given. You can terminate a variable watch by using the "V" command with no arguments. POD will automatically terminate a watch on a variable when that variable is no longer available. When POD does this, it prints the message "Watch terminated -- value didn't change".

```
} B(EVALUATEBOARD,35); C
Breakpoint at EVALUATEBOARD,35  FOR I:=5 TO 39 DO
} V(BLACKSCORE)<W(WHITESCORE)>
} C
Value of "BLACKSCORE" changed at statement:
EVALUATEBOARD,224  ELSE BLACKSCORE:=BLACKSCORE+MOC4;
Old value: 0
New value: 400
Breakpoint at EVALUATEBOARD,225  IF BLACKDENY<WHITEDENY THEN
0
} C
Watch terminated -- value didn't change
Breakpoint at MAIN,28  MAXLEVEL:=0;
}
```

W(): Write Variable Value

The "W" command is used to write the value of a variable, pointer, constant, or memory location. The format of the output is determined by the type of the variable being written. For example, integer variables are written as 16-bit signed decimal integers, while set variables are written with set notation. The names of the variable to be displayed are placed inside parentheses after the "W". If more than one variable is to be written then the names are separated by commas. Physical memory locations are addressed as integers (either octal or decimal). As in Pascal, integer and real values may use format control with the colon (:) notation. This is also how one examines memory locations in octal.

```
} W(TURN)
BLACK
} W(COLOR[BLACKKING],COLOR[WHITEKING])
BLACK
WHITE
} W(USERMOVES[5])
'B1'
} W(ROOT^.SON^.VALUE)
402
} W(54B)
-10154
} W(54B:-1)
154126B
} W(S)
['A','M','Z']
} W(R)
 3.141593E+00
} W(CH)
'A'
} W(I)
123
}
```

Advanced Debugging Techniques

If you write large programs, restrictions in memory size may limit your use of POD for debugging. However, you can do several things to reduce the amount of memory required by POD.

The easiest thing is to disable source debugging. The use of the source debugging option (/S) expands your program by one word for every Pascal statement in your program. For large programs, you may save more than 1K words by not using source debugging.

Another technique you can use is selective debugging. You can edit your program to turn off the generation of POD debugging information around procedures that have already been tested and debugged. To turn off debugging, place the line {$D-} before the procedure definition and {$D+} after the procedure. You will not be able to set breakpoints or examine variables in such procedures, but you will save two or three words for every statement not debugged. Be sure debugging is enabled around all variables you may wish to examine and around the main procedure.

If your program uses overlays, you can still debug your program using POD. When you compile the main body of the program, which resides in the root segment, use the debugging switch (/D) and produce a symbol table file. Compile each of the external modules in the normal way without the debugging switch.

The best way to debug an external procedure is to edit your main program to include the definition of the external procedure. Once the procedure has been debugged, you can move the procedure from the main program into an external module.

Another way to debug one external procedure at a time is to compile that external procedure with the debugging option (/D). Edit your main program to enable debugging only around the BEGIN statement that is the start of the main program as shown:

```
{$D+}
      BEGIN {MAIN}
{$D-}
```

This initializes the Debugger when the program starts. Then compile the main program in the nromal way. When POD asks you for the program name, give the name of the external procedure you wish to debug.

When you link your overlaid program you will have to use two overlay regions to contain the modules of POD. These two overlay regions may, in most cases, also contain your own external procedures. There should be no conflicts because POD only lets you debug in the root segment, and as long as the two POD modules RTDBG and DBG are placed in the root, there should be no problems with the overlays.

You cannot set breakpoints within external procedures, but you can cause a break when the external procedure is called from the main program. This is done by setting a breakpoint and giving only the name of the procedure at which to break as with: B(OVER1). This type of breakpoint will stop the program before the external procedure OVER1 is executed. The only variables you will be able to examine and modify in OVER1 are those variables in the parameter list for OVER1. Note that the names of the parameters are defined by the external procedure definition of OVER1 in the main program, not by the definitions in OVER1 itself.

SECTION 5: INSTALLATION GUIDE

Introduction to the Installation Guide

This guide describes the procedures for installing Pascal-1 V1.2 on your RT-11 operating system. Note that the procedures for the installation of V2 of RT-11 differ from those for V3 and V4 of RT-11. The latter two are identical.

In examples, underlining is used to show the text that you should type. Non-underlined text shows the prompts or other responses by the computer.

Contents of the Distribution Medium

Each medium contains a complete set of distribution files. Below is a list of these files with a brief explanation of each one.

Installation Files

INSTAL.DOC	Installation documentation
INSTAL.SAV	Installation program

Documentation Files

TABLE .DOC	Table of Contents
INTRO .DOC	Introduction
USER .DOC	User's Guide .
GUIDE .DOC	Programmer's Guide
PASCAL.DOC	Language Specification
DEBUG .DOC	Debugger Guide
INSTAL.DOC	Installation Guide

Compilers

PASFPP.SAV	Compiler for FPP machines
PASSIM.SAV	Compiler for non-FPP machines

Utility Programs

PASFMT.PAS	Formatter and cross reference
IMP .PAS	Post-compilation optimizer
PROFIL.PAS	Program profiler
PCL .PAS	Pascal command language interpreter
ERROR .PAS	System Error() procedure
STRING.PAS	String manipulation procedures
INTRPT.PAS	Interrupt handling examples
CSI .PAS	Interface for .CSIGEN
MACEIS.SAV	Fast assembler for EIS machines
MACSIM.SAV	Fast assembler for non-EIS machines

Object Libraries

LIBFPP.V3/.V2	Library for FPP on RT-11 V3/V2
LIBFIS.V3/.V2	Library for FIS on RT-11 V3/V2
LIBEIS.V3/.V2	Library for EIS on RT-11 V3/V2
LIBSIM.V3/.V2	Library for non-EIS on RT-11 V3/V2

Debugger Modules

```
0FPP.V3/.V2, ... ,9FPP.V3/.V2,AFPP.V3/.V2,BFPP.V3/.V2
        FPP debugger modules for RT-11 V3/V2
0SIM.V3/.V2, ... ,9SIM.V3/.V2,ASIM.V3/.V2,BSIM.V3/.V2
        Non-FPP debugger modules for RT-11 V3/V2
```

Demonstration Programs

```
HEARTS.PAS  The game of Hearts
RANDOM.PAS  Random number generator
MAZE  .PAS  Amazing Demonstration
```

Installation Preparation

For RT-11 V3 and V4, the Pascal V1.2 system is installed by an automatic configuration procedure. For RT-11 V2, the procedure is only partially automated.

Several system programs supplied by DEC are required for installation of Pascal and for use by the Pascal system. These programs are PIP.SAV, the file transfer utility; MACRO.SAV, the assembler; SYSMAC.SML, the MACRO library; SYSLIB.OBJ, the default system object library; and LINK.SAV, the linking loader. You should verify that these programs are on your system disk.

Installation Dialogue

The installation procedure builds Pascal on a single target disk. The target disk must contain at least 544 free blocks if no documentation or demonstrations are desired. The documentation files require 245 blocks of storage, while the demonstration files use 175 blocks, or 964 total blocks.

To start the installation process, type:

```
.RUN xxn:INSTAL
```

where xxn: is the device name and unit number of the distribution medium. This starts the installation procedure, which will ask several questions. The default answer is given in parentheses at the end of each question and will be used when you respond with only a carrage return. Below are the questions and a description of each.

DEVICE FOR THE DISTRIBUTION MEDIUM (MT:) ?

Enter the device name and unit number on which the Pascal distribution medium is mounted. If the device is "MT:", a carriage return will use this name as default.

DEVICE FOR THE TARGET DISK (SY:) ?

Enter the device name and unit number of the disk on which Pascal is to be installed.

INCLUDE DEMONSTRATION PROGRAMS AND DOCUMENTATION (YES) ?

If the answer is yes, the demonstration programs and documentation files are included in the system installation procedure.

USE THE CURRENT SYSTEM'S CONFIGURATION (YES) ?

If the answer is yes, the version of RT-11 and the math hardware on the current system are used in configuring Pascal, and the following questions are omitted.

VERSION OF RT-11 TO BE USED (n) ?

Answer "4" if Pascal is to run on RT-11 V4. Answer "3" if Pascal is to run on RT-11 V3. Answer "2" if Pascal is to run on RT-11 V2. The default will be the current version of RT-11. The same Pascal system is used for RT-11 V3 or V4.

IS FPP HARDWARE TO BE USED (YES/NO) ?

If you will be using a machine with an FP11 floating-point

processor, answer yes, otherwise no. If yes, then the next two questions are skipped. The default will be the current system's configuration.

IS FIS HARDWARE TO BE USED (YES/NO) ?

If you will be using a machine with a PDP-11/40 style floating-point instruction set answer yes, otherwise no. If yes, then the next question is skipped. The default will be the current system's configuration.

IS EIS HARDWARE TO BE USED (YES/NO) ?

If you will be using a machine with an extended instruction set for integer operations answer yes, otherwise no. The default will be the current system's configuration.

After these questions have been answered, the configuration to be installed will be printed, as in the example:

```
CONFIGURATION BEING INSTALLED
    SOURCE:   MT:
    TARGET:   RK2:
    RT-11:    V3
    MATH:     EIS
```

The installation program will then proceed to copy the files from the distribution medium to the target device.

For floppy and DECtape distribution, the installation program will prompt the user to mount the next volume when needed.

After all files have been copied, several programs need to be compiled. These are PASFMT.PAS, the Pascal text formatter; IMP.PAS, the macro code improver; PCL.PAS, the Pascal command interpreter; and PROFIL.PAS, the Pascal program profiler object module.

For RT-11 V3 and V4, an indirect command file automatically compiles these programs. See Appendix A for a sample installation. You also will need to customize the Pascal command language program, PCL, as described in Appendix C, if the target device is other than the system disk.

Installation on V2 or Floppy Disks

For RT-11 V2 or for floppy drive systems, you must compile the above programs by typing some or all of the commands on the console terminal, following the same general procedure shown in Appendix A.

Memory storage is a problem with floppys, so your first action is to clean your disk. The next step depends on whether you have double-density drives or single-density drives as main storage.

Double-Density Drives

If your system has double-density drives with at least 544 free blocks, you may use the automated installation driver, INSTAL.SAV, to partially build your Pascal system. This means you can follow "Step 1: Automatic Installation" described in the sample installation in Appendix A. The installation procedure will run out of space shortly after creating the indirect command file, PASCAL.COM. The last message you normally will see before the crash is:

```
All files have been copied -- beginning compilations.
```

You will have to manually perform all of the steps that follow this message, under the heading "Step 2: Compilation Process" shown in Appendix A. You will have to use an additional drive for intermediate files. All files used by Pascal-1 V1.2 can be stored on a system disk.

V2 or Single-Density Drives

If you have RT-11 V2 or if your system uses single-density drives, then you must build Pascal-1 manually. Begin with these steps:

1) Select a compiler from PASFPP.SAV or PASSIM.SAV and call it PASCAL.SAV.

2) Select a support library from LIBFPP.V2/V3, LIBFIS.V2/V3, LIBEIS.V2/V3, or LIBSIM.V2/V3 and call it PASCAL.OBJ.

3) Select the debugger modules. The debugger consists of 12 object modules that must be copied onto your system as A.OBJ, B.OBJ, 0.OBJ,...,9.OBJ. The distribution kit contains debugger modules for RT-11 versions 2 and 3/4 supporting FPP and FIS/EIS/SIM floating-point hardware. Choose those 12 files appropriate to your system.

After transferring the files given above, you have a Pascal system capable of compiling, assembling, linking, debugging, and executing Pascal programs.

You may now generate the utilities PASFMT, IMP, PCL, PROFIL, and STRING. You may then store these utilities on another device to save space. You also may delete the intermediate .MAC and .OBJ files that were created in the installation process, then "squeeze" your disk to reclaim space. You must, however, save PASCAL.OBJ, the 12 debugger modules and PROFIL.OBJ. After cleaning, then, device SY: should have these files:

```
PASCAL.SAV      (Compiler)
PASCAL.OBJ      (Run-time support)
A,B,0-9.OBJ     (Debugger modules)
PCL.SAV         (Command Language Interpreter)
IMP.SAV         (Macro code improver)
PASFMT.SAV      (Source text formatter)
PROFIL.OBJ      (Profiler module)
MAC.SAV         (Fast assembler)
   (plus RT-11 system utilities and files)
```

This sequence will bring you up to "Step 2: Compilation Process" described in Appendix A; you should manually complete all of the commands in Step 2.

Sample Installation

Step 1: Automatic Installation

This installation is from magtape to the system disk on RT-11 V3 with FPP. The "<CR>" means that a carriage return is the only response.

```
.RUN MT:INSTAL
INSTAL --- System installation for OMSI Pascal-1 V1.2 on RT-11

For the following questions, a default answer is
given in parentheses.  The default will be used when
you press the carriage return key. For details about the
question, type a '?';  otherwise type the desired response.

Device for the distribution medium (MT:) ? <CR>

Device for the target disk (SY:) ? <CR>

Include demonstration programs and documentation (YES) ? <CR>

Use current system's configuration (YES) ? <CR>

        Configuration being installed:
                SOURCE: MT:
                TARGET: SY:
                RT11:   V3
                MATH:   FPP

Copying files:
        MT:USER  .DOC  to  SY:USER  .DOC
        MT:PASCAL.DOC  to  SY:PASCAL.DOC
        MT:DEBUG .DOC  to  SY:DEBUG .DOC
        MT:GUIDE .DOC  to  SY:GUIDE .DOC
        MT:PASFPP.SAV  to  SY:PASCAL.SAV
        MT:MACEIS.SAV  to  SY:MAC   .SAV
        MT:LIBFPP.OBJ  to  SY:PASCAL.OBJ
        MT:AFPP  .V3   to  SY:A     .OBJ
        MT:BFPP  .V3   to  SY:B     .OBJ
        MT:0FPP  .V3   to  SY:0     .OBJ
        MT:1FPP  .V3   to  SY:1     .OBJ
        MT:2FPP  .V3   to  SY:2     .OBJ
        MT:3FPP  .V3   to  SY:3     .OBJ
        MT:4FPP  .V3   to  SY:4     .OBJ
        MT:5FPP  .V3   to  SY:5     .OBJ
        MT:6FPP  .V3   to  SY:6     .OBJ
        MT:7FPP  .V3   to  SY:7     .OBJ
        MT:8FPP  .V3   to  SY:8     .OBJ
        MT:9FPP  .V3   to  SY:9     .OBJ
        MT:PASFMT.PAS  to  SY:PASFMT.PAS
        MT:IMP   .PAS  to  SY:IMP   .PAS
        MT:PROFIL.PAS  to  SY:PROFIL.PAS
```

```
MT:PCL   .PAS  to  SY:PCL   .PAS
MT:STRING.PAS  to  SY:STRING.PAS
MT:HEARTS.PAS  to  SY:HEARTS.PAS
MT:CHECKR.PAS  to  SY:CHECKR.PAS
MT:RANDOM.PAS  to  SY:RANDOM.PAS
MT:MAZE  .PAS  to  SY:MAZE  .PAS

   All files have been copied -- beginning compilations
```

Step 2: Compilation Process

After copying all these files, the automatic installation program creates the indirect command file PASCAL.COM, which then compiles the Pascal source programs. The compilation process is not shown here for reasons of brevity.

For RT-11 V2 users and for double-density and single-density floppys, this command file will not be created; the user must do the following steps manually to compile the supplied source programs.

```
.ASS yyn DK                    (yyn: is the target device)
.RU PASCAL
*PASFMT=PASFMT
.RU PASCAL
*IMP=IMP
.RU PASCAL
*PROFIL=PROFIL
.RU PASCAL
*PCL=STRING,PCL
.R MACRO
*PASFMT=PASFMT
*IMP=IMP
*PROFIL=PROFIL
*PCL=STRING,PCL
*^C
.R LINK
*PASFMT=PASFMT,PASCAL
*IMP=IMP,PASCAL
*PCL=PCL,PASCAL/M:60000
*^C
.R PIP
*PASFMT.PAS,PASFMT.MAC,PASFMT.OBJ,IMP.PAS,IMP.MAC,IMP.OBJ/D
*PROFIL.PAS,PROFIL.MAC,PCL.MAC,PCL.OBJ/D
*^C
```

and if the demonstration programs are included:

```
.RU PASCAL
*HEARTS=HEARTS
.RU PASCAL
*CHECKR=CHECKR
```

```
.RU PASCAL
*MAZE=MAZE
.R MACRO
*HEARTS=HEARTS
*CHECKR=CHECKR
*MAZE=MAZE
*^C
.R LINK
*HEARTS=HEARTS,PASCAL
*CHECKR=CHECKR,PASCAL
*MAZE=MAZE,PASCAL
*^C
.R PIP
*HEARTS.MAC,HEARTS.OBJ,CHECKR.MAC,CHECKR.OBJ/D
*MAZE.MAC,MAZE.OBJ/D
*^C
```

Programming Changes in Pascal-1 V1.2

Four specific language features have been changed from V1.1 to V1.2, all of which are related to I/O characteristics. If a program that was written in V1.1 fails to operate properly with V1.2, check these points:

(1) In V1.1, Eoln() on interactive terminal files had the initial value True. This has changed to False in V1.2.

Symptom: Program "hangs", or ignores the first input line.

Cure: Remove the initial Readln(), or replace it by the statement "if Eoln() then Readln()", which runs correctly with either version.

(2) The V1.1 Read() procedure, when reading a (packed) array of Char, ignored leading blanks and terminated on a blank or a comma. The V1.2 Read() procedure will read characters without skipping blanks, and terminates at Eoln() or when the array is filled.

Symptom: Program does not interpret commands properly, or loops.

Cure: Reprogram sections that use the V1.1 Read(). Programs that are heavily dependent on the V1.1 style Read() may be more easily recoded with the V1.2 string package.

(3) The declaration "file of Char" is no longer equivalent to the declaration "Text". This change corresponds to the more strict type checking of the draft ISO Pascal Standard.

Symptom: Compiler error message "TEXT file expected".

Cure: Substitute the type Text and recompile.

(4) The Seek(), Deposit(), and CloseRandomFile() procedures supplied with V1.1 have been superseded by the built-in procedure Seek().

Symptom: Unpredictable I/O failures. The V1.1 procedures will compile under V1.2, but will not operate correctly.

Cure: Reprogram affected sections using the built-in Seek(). Note that the V1.2 Seek() numbers file records beginning at 1.

Customizing the Pascal Language Command Program (PCL)

For sites using RT-11 V3 or V4, the Pascal command language program PCL can be used to automate the compilation process. PCL accepts a line of input in standard Command String Interpreter (CSI) format and produces an indirect command file to perform the compilation, assembly, linking, and start-up of a program.

When Pascal-1 is installed on the system disk, PCL will need no modification; the PCL.SAV created in the installation procedure will work correctly.

If Pascal-1 is to be on a non-system disk, PCL.PAS will need modification and recompilation. The editing can be done with a standard text editor. Change the first line of PCL.PAS to:

```
PascalDevice = 'yyn:';  (* Device for the Pascal files *)
```

where yyn: is the device name and unit number on which the Pascal files can be found. After this change has been made, recompile PCL with the following commands:

```
.ASSIGN yyn DK
.R PASCAL
*PCL=PCL
.MAC PCL
.LINK PCL,PASCAL/STACK:60000
```

Note: PCL features are described in detail in a comments section at the beginning of the PCL.PAS program.

```
PROGRAM convertgroups (datafile, binaryfile, output);
CONST
groupsize = 10;
TYPE
index = 1 .. groupsize;
group = ARRAY [index] OF real;
VAR
datafile : text;
binaryfile : FILE OF group;
ix : index;
groupcount : integer;
BEGIN
reset(datafile);
rewrite(binaryfile);
groupcount := 0;
REPEAT
ix :=1;
read(datafile,binaryfile^[ix]);
IF NOT eof(datafile)
THEN
BEGIN
REPEAT
ix := ix + 1;
read(datafile,binaryfile^[ix])
UNTIL (ix = groupsize) OR eof(datafile);
IF eof(datafile)
THEN writeln('File ends with short group')
ELSE
BEGIN
put(binaryfile);
groupcount := groupcount + 1
END
END
UNTIL eof(datafile);
writeln(groupcount, 'Groups converted')
END. { convertgroups }
```

```
VAR     I:INTEGER;

BEGIN
        WRITELN;
        CASE CLASS OF
            WARNING: WRITE('Warning: ');
            IOERROR: WRITE('?I/O error: ');
            ELSE WRITE('?Fatal error: ');
        END;
        writeln(msg:errormsglength);
        IF CLASS=IOERROR THEN BEGIN
           IF FILENAMELENGTH > 0 THEN
               WRITELN(' Filename: "',filename:filenamelength,'"');
              writeln(' I/O status: ',IOSTATUS:1);
        END;
        WRITELN(' Program counter: ',USERPC:-1);
END;
```